ISSN 0265-2919

90p 82

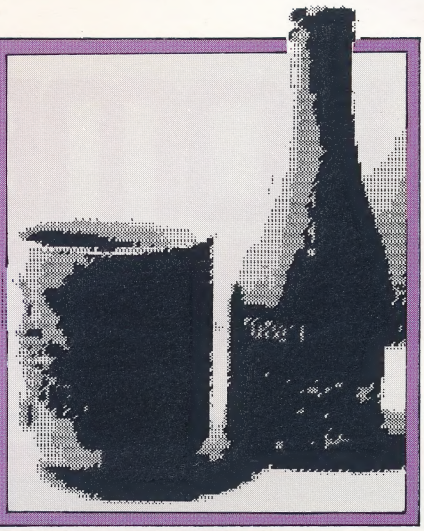
THE HOME COMPUTER ADVANCED COURSE

MAKING THE MOST OF YOUR MICRO

IR£1.15 Aus $2.15 NZ $2.65 SA R2.45 Sing $4.50

CONTENTS

Next Week



- **Video digitisers allow a video signal to be converted into a form that can be processed by a computer. We discuss the principles behind these devices and look at the Print Technik Video Digitiser for the Commodore 64.**
- **Probability theory is at the heart of the scientific approach to gambling. We consider the ways in which these mathematics have been implemented on micros.**
- **We conclude our FORTRAN series by discussing ways in which the language's limitations can be overcome.**

QUIZ

1) What does the SOUND_HOLD entry do in the Amstrad's operating system?
2) How much memory is free for applications on the Amstrad 6128 once CP/M Plus has been installed?
3) What is the major problem encountered in master-slave systems?

Answers To Last Week's Quiz
1) The Tab key.
2) The up arrow, which indicates exponentiation.
3) Z5 Horse Race Forecast.
4) Ink previously would block the chambers leading to the jet nozzle; Epson's new ink delivery system claims to avoid this problem.

Coming Up

- A series of articles investigating the wide range of careers involving computers.
- A short programming techniques series that discusses, among other things, sorting techniques and text compression.

Editor Stephen Cooke; **Art Editor** Claudia Zeff; **Deputy Editor** Steve Colwill; **Production Editor** Bobby Pickering; **Designer** Julian Dorr; **Staff Writer** Steve Malone; **Art Assistant** Caroline Clayton; **Sub Editor** Jon Kaye; **Contributors** Chris Honey, Chris Laing, Dougie Bern, Tony Drapkin, Pete Connor, Mike Curtis, Steve Cooke, Steve Colwill, Steve Malone, Martin Young; **Software Consultants** Pilot Software City; **Group Art Director** Perry Neville; **Managing Director** Stephen England; **Published by Orbis Publishing Ltd: Editorial Director** Brian Innes; **Project Development** Peter Brookesmith; **Executive Editor** Maurice Geller; **Production Assistant** Susan Brown; **Subscription Manager** Christine Allen; **Designed and produced by Bunch Partworks Ltd; Editorial Office** 14 Rathbone Place, London W1P 1DE; **© APSIF Copenhagen 1985; © Orbis Publishing Ltd 1985: Typeset by Universe; Reproduction by Mullis Morgan Ltd; Printed in Great Britain by Heanor Gate Printing Ltd, Derby**

HOW TO OBTAIN ISSUES AND BINDERS FOR THE HOME COMPUTER ADVANCED COURSE – Issues can be obtained by placing an order with your newsagent or direct from our subscription department. If you have any difficulty obtaining any back issues from your newsagent, please write to us stating the issue(s) required and enclosing a cheque for the cover price of the issue(s). **AUSTRALIA** – please write to: Gordon & Gotch (Aus) Ltd, 114 William Street, PO Box 767G, Melbourne, Victoria 3001. **MALTA, NEW ZEALAND & SOUTH AFRICA** – Back numbers are available at cover price from your newsagent. In case of difficulty, write to the address given for binders.
UK/EIRE – Issue Price: 90p/IR£1.15. Subscription: 6 months: £26.00. 1 Year: £52.00. Binder: please send £3.95 per binder, or take advantage of our special offer in early issues. **EUROPE** – Issue Price: 90p. Subscription: 6 months air: £44.72. Surface: £36.14. 1 year air: £89.44. Surface: £72.28. Binder: £5.00. Airmail: £8.25. **MALTA** – Obtain binders from your newsagent or Miller (Malta) Ltd, MA Vassalli Street, Valetta, Malta. Price: £3.95. **MIDDLE EAST** – Issue Price: 90p. Subscription: 6 months air: £50.18. Surface: £36.14. 1 year air: £100.36. Surface: £72.28. Binder: £5.00. Airmail: £8.25. **AMERICAS/ASIA/AFRICA** – Issue Price: US/CAN$1.95/90p. Subscription: 6 months air: £59.54. Surface: £36.14. 1 year air: £119.08. Surface: £72.28. Binder: £5.00. Airmail: £9.50. **SOUTH AFRICA** – Issue Price: SA R2.45. Obtain binders from any branch of Central News Agency or Intermag, PO Box 57394, Springfield 2137. **SINGAPORE** – Issue Price: Sing $4.50. Obtain binders from MPH Distributors, 601 Sims Drive, 03-07-21, Singapore 1438. **AUSTRALASIA/FAR EAST** – Issue Price: 90p. Subscription: 6 months air: £64.22. Surface: £36.14. 1 year air: £128.44. Surface: £72.28. Binder: £5.00. Airmail: £9.75. **AUSTRALIA** – Issue Price: Aus$2.15. Obtain binders from First Post Pty Ltd, 23 Chandos Street, St Leonards, NSW 2065. **NEW ZEALAND** – Issue Price: NZ$2.65. Obtain binders from your newsagent or Gordon & Gotch (NZ) Ltd, PO Box 1595, Wellington.
ADDRESS FOR BINDERS AND BACK ISSUES – Orbis Publishing Limited, Orbis House, Bedfordbury, London WC2 4BT. Telephone 01-379 5211. Cheques/postal orders should be made payable to Orbis Publishing Limited. Binder prices include postage and packing and prices are in sterling. Back issues are sold at the cover price, and we do not charge carriage in the UK.
NOTE – Binders and back issues are obtainable subject to availability of stocks. Whilst every attempt is made to keep the price of the issues and binders constant, the publishers reserve the right to increase the stated prices at any time when circumstances dictate. Binders depicted in this publication are those produced for the UK and Australian markets only. Binders and issues may be subject to import duty and/or local taxes, which are not included in the above prices unless stated.
ADDRESS FOR SUBSCRIPTIONS – Orbis Publishing Limited, Hurst Farm, Baydon Road, Lambourn Woodlands, Newbury Berks, RG16 7TW. Telephone: 0488-72666. All cheques/postal orders should be made payable to Orbis Publishing Limited. Postage and packaging is included in subscription rates, and prices are given in sterling.

COVER ILLUSTRATION BY ALAN ADLER

COURTESY OF MECCA BOOKMAKERS

Window Display
Computer-enhanced information displays are just one of the many uses now being made of the new technology by the larger betting chains. Others include monitoring customer activity, staff productivity and calculating odds on special betting options

ODDS-ON FAVOURITES

While many of us might bet on a horse whose name somehow strikes our fancy, professional bookmakers and punters look more to statistics and the science of chance and probability. Programs written specifically for race forecasting must therefore take into account many factors.

Bookmakers could almost be regarded as middlemen who provide a service, enabling punters with different opinions about the merits of the horses in a given race to back their judgements with hard cash. The odds the bookmakers offer reflect the weight of money staked — the more people back a horse, the more money goes on to it and therefore the 'shorter' its price. In addition, bookmakers attempt to juggle the odds so that, in general, they are certain to take out a percentage of the total whichever horse wins. In this way, punters are effectively charged for bookmaking services.

The betting market really takes off at the racecourse in the 10 minutes or so of frantic activity immediately preceding the race. Betting 'shows' are relayed to betting shops throughout the country so that millions of off-course punters can have a bet.

Despite the obstacles to the introduction of new technology on-course, some off-course bookmakers (the large chains in particular) are increasingly turning to computers to boost efficiency and profitability. Ladbrokes, for example, uses computers extensively in the credit betting side of the business. All credit clients' bets are recorded on Cromemco micros connected to a hard disk. At the end of the day, the data is transferred to tape, which is fed into an IBM mainframe that has already been supplied with the day's results from all meetings. All bets, whether straight wins, each-way, or the various combinations and accumulators, are then settled automatically. Clients' accounts are updated weekly, and cheques issued or requests for payment made. In addition, the mainframe is used for the standard data processing tasks to be found in any large business.

As the major firms come to think of themselves less as bookmakers and more as a kind of leisure industry, there is an increasing tendency to use computers in other roles. Mecca Bookmakers, for example, is confident that the falling cost of computing power will pave the way to a much more sophisticated analysis of their business. Terminals placed in selected branches could be used to conduct a thorough investigation of the punters' preferences, making it possible to identify, for example, the most popular kind of bet, the facilities that people would like to see provided, or the kind of manager conducive to an increase in business.

Mecca believes that computer-aided market research will grow in importance as new legislation finally allows betting shops to become more

congenial and welcoming places to visit. The company is already using its mainframe to enhance graphic displays of information on racing and other sporting events, transmitting them in a bright, colourful form to selected offices in the chain. In addition, computers are helping to monitor the performance and profitability of individual shops and the group as a whole, ensuring that management knows clearly whether or not it's achieving its projected targets.

Most off-course bookmakers now offer a chance to predict the first two or three runners in correct order. These bets are known respectively as the Computer Straight Forecast (CSF) and Computer Tricast. The payout on both of these bets is calculated by computer according to complex, previously established formulae. In the early days of horse race forecasts, some bookmakers simply multiplied the odds in order to calculate winning bets. It was soon realised, however, that their dependence on the vagaries of course odds left them with inadequate margins in the field of forecast betting, particularly when, as frequently happens, the favourite horses finished first and second.

THE PAYOUT FORMULA

A formula was devised to determine the payout on every possible combination of odds such that the bookmakers were guaranteed a minimum margin of 20 per cent. In 1977, the formula was computerised and the CSF was born, followed in 1981 by the Tricast. The performance of the formula is constantly monitored by the Computer Technical Sub-Committee of the Betting Offices Licensees' Association (BOLA) and the National Sporting League (NSL), representing the interests of large and medium-sized chains, respectively. Amendments to the original program (written in BASIC) are made in response to changes in betting patterns or starting price returns.

The most celebrated alteration occurred following what became known as the 'Little Owl affair'. In January 1982, a three-horse race over fences involved a hot favourite (Little Owl) at 11-4 'on' (i.e. 4-11), a second favourite at 5-2 and a rank outsider at 66-1. For some reason, Little Owl was pulled up early in the race leaving the 5-2 shot to come home ahead of the outsider. Some bookmakers took a hammering. Others hinted darkly that they had been victims of a professional foul. The resultant placing of a patch (the 'harmonic factor') on the original program ensured that in small fields producing surprise results, the odds of rank outsiders are depressed for forecast purposes while those of more fancied horses are extended slightly. Thus, under the old formula, the Little Owl forecast would have paid 141-1. Under the revised formula, this would be reduced to a mere 14-1.

The computer therefore provides the large off-course bookmaker with a powerful tool for analysing his business, identifying areas of vulnerability, and protecting and enhancing his

On-Course To Win

On-course bookmakers have, over the years, developed a complex system of interaction and communication designed to meet market demands and protect themselves against financial loss. On-course bookmakers may be divided into two classes. First, there are the larger bookmakers (or 'rails' bookmakers, so called because of the position they occupy by the rails dividing the members' enclosure from the public), who take bets from accredited clients only and originate the market. Then there are the bookmakers in the public arena (Tattersalls), who take bets in cash from members of the public.

In the space of a few hectic minutes before the race, the betting market explodes into a great surge of activity, in which tens of thousands of pounds change hands in thousands of separate transactions. During this period, bookies communicate with each other via a 'Tic-Tac Man', who relays messages between parties using a complex code of hand signals.

One of the principal functions of the Tic-Tac Man is to facilitate the process of 'laying off' bets. When a bookie becomes aware that he has laid himself open to possible heavy losses on a particular horse, he will effectively 'pass the buck' by placing, via the Tic-Tac Man, a bet on that particular horse with another bookie. In this way, if the horse in fact wins, he will be able to recoup some of his losses.

It seems unlikely that the computer will ever impinge upon such time-honoured practices for the simple reason that everything happens too fast. It would be comparatively easy to write a program that simulated the bookmaker's activities, but data input would be far too slow to cope with the final rush of betting that takes place for each race. In the Tic-Tac Man, it would seem, the computer may have met its match

all-important margins.

As far as the punter is concerned, one fact is inescapable. No program designed for use with a home micro is going to make the user rich overnight, or drive the bookmakers out of business. On learning this, most would-be computer punters might become discouraged but the micro can help the discerning backer in a number of ways, which deserve to be examined.

Although it is true that punters in general must lose — the bookmaker's margin and betting tax see to that — it would be wrong to assume that they all do so. The shrewdest punters — those with a superior ability to read the form and weigh up the odds — win, and their winnings come from those who are less able than themselves. The thoughtful punter devotes considerable time to the analysis of the racecourse performances of horses in order to establish a clear picture of their relative merits.

If the field can be narrowed down to two or three with an obvious chance, then the race is worth considering. Finding one horse with an outstanding chance is difficult, but occasionally possible. Further factors such as the horse's fitness, the conditions of the race (distance, going, jockey and so on) and, crucially, the betting, must then be considered. Value for money is, of course, essential — finding a horse with an obvious chance at good odds. This is not easy, though, since after all, if the horse's chance is blindingly obvious, even the most incompetent punter will take advantage of the opportunity, thus ensuring that the horse starts at cramped odds.

The successful punter's task, therefore, is to find a horse with an overwhelming chance that does not attract a vast flood of money, and which is consequently likely to start at longer odds than it should. Such betting opportunities take time to identify and are only rarely to be found, so that the principle of rigorous selectivity is central to the successful punter's strategy.

There simply isn't time to assess the form in every one of the minimum six races per meeting. Given that betting shop punters have access to between two and five meetings per afternoon, it is scarcely surprising that the vast majority operate on a consideration of form which is little better than crude guesswork. Of course, sometimes this approach works, but the overall trend is inexorably towards the accumulation of losses.

It's here that programs for the gambler (see page 1601) enter the scene. The virtue of these is that they impose a degree of selectivity on the punter and thus inhibit the undisciplined approach, which is the downfall of so many. The programs do this because of what some might regard as one of their principal drawbacks — namely, the slowness of data input, which can take half an hour or so per race by the time all the relevant variables have been input. Clearly, the punter has to decide in principle to consider only those races which offer the best chances. One strategy that has much to recommend it is generally to bet in races of ten runners or less, thus cutting down input time and simultaneously increasing your chances of winning.

The computer punter is unlikely, however, to strike terror into the heart of the bookmaker. The fact is that such programs only appeal to the more thoughtful and selective backer, from whose point of view it is probably better that this state of affairs should continue. Any radical change in betting patterns will be rapidly identified by the bookmaking fraternity, and market forces will inevitably combine to ensure that, whatever else happens, the 'satchel man' gets his percentage at the end of the day.

Flat Chance
This table is typical of those frequently published by The Sporting Life. It gives an analysis of the performance of a particular trainer (in this case David Elsworth) during the year indicated. The figures are broken down under various headings likely to be of interest to the punter. Such analyses would be impossibly time-consuming without the computer and have only become available during the past few years

1984 Flat statistics

	No. of Horses	Races Run	1st	2nd	3rd	Unpl	Per cent
2-y-o	15	44	2	4	8	30	4.5
3-y-o	25	119	16	7	13	83	13.4
4-y-o+	14	51	6	5	4	36	11.8
Totals	54	214	24	16	25	149	11.2

	£1 Level Stake
2-y-o	- 14.00
3-y-o	- 19.90
4-y-o+	+ 3.33
	- 30.57

Jockeys Riding	W-R	£1 Level Stake
B Rouse	11-94	- 23.92
S Cauthen	6-20	+ 33.25
D Brown	2-18	+ 1.00
R Fox	2-20	+ 10.00
Pat Eddery	0-3	
G Duffield	0-2	
J H Brown	0-1	
Paul Eddery	0-1	
J Mercer	0-1	
W Carson	0-1	
D Gibson	0-1	
T Ives	0-1	

	Mar/Apr	May	Jun	Jul	Aug	Sep	Oct/Nov
2-y-o	0-0	0-3	1-9	0-4	0-7	0-10	1-11
3-y-o	0-6	4-24	1-19	4-21	1-17	2-13	4-19
4-y-o+	2-6	0-7	1-9	0-5	2-9	1-7	0-8
Totals	2-12	4-34	3-37	4-30	3-33	3-30	5-38

	W-R	£1 Level Stake
Non-Handicaps		
2-y-o	2-37	- 7.00
3-y-o	8-77	- 22.65
4-y-o+	0-4	- 4.00
Selling	2-8	- 0.17
Apprentice	1-1	+ 7.00
Amateur/Ladies	0-0	0.00
Totals	13-127	- 26.82

Handicaps	W-R	£1 Level Stake
2-y-o	0-3	- 3.00
3-y-o	5-36	- 10.25
4-y-o+	4-40	+ 2.00
Selling	1-3	+ 1.50
Apprentice	1-2	+ 9.00
Amateur/Ladies	0-3	- 3.00
Totals	11-87	- 3.75

Course Record	£1 Level Stake
Brighton	
Folkestone	
Windsor	
Salisbury	
Ascot	
Redcar	
Warwick	
York	
Bath	
Lingfield	
Kempton	
Newmarket	
Goodwood	
Newbury	
Sandown	
Chepstow	
Ayr	

Non-handicaps 2-y-o	3-y-o+	Handicaps 2-y-o	3-y-o+
0-2	3-10	0-0	1-1
1-4	1-4	0-0	1-5
0-7	2-9	0-0	1-11
1-6	0-10	0-1	2-14
0-4	1-3	0-1	1-12
0-0	1-1	0-0	0-1
0-0	0-0	0-0	1-2
0-0	0-0	0-0	1-2
0-2	0-4	0-0	1-3
0-4	1-5	0-0	0-1
0-2	0-3	0-0	1-6
0-2	0-4	0-0	0-5
0-1	1-8	0-0	0-2
0-1	1-11	0-1	

	W-R	£1 Level Stake
P Cook	1-4	- 1.90
L Piggott	1-6	- 1.50
A McGlone	1-24	- 19.50

R P Elliott	0-2	0-2
R Cochrane	0-1	0-1
E Hide	0-1	0-1
J Matthias	0-1	0-1
B Jago	0-1	0-1
Mr P Thomas	0-1	0-1
M L Thomas	0-1	

	Total W-R	£1 Level Stake
2-y-o	4-13	+ 14.25
3-y-o	3-13	+ 11.58
4-y-o+	3-27	+ 0.50
Selling	3-31	- 7.50
Apprentice	2-20	+ 7.00
Amateur/Ladies	1-2	+ 0.10
	1-2	+ 2.50
	1-2	+ 7.00
	1-9	- 2.00
	1-10	- 7.50
	1-11	+ 4.00
	1-12	- 7.50
	1-14	- 10.50
	1-15	- 9.50

Course Grade	W-R	%	£1 Level Stake
Group 1	7-100	7.0	- 35.50
Group 2	9-56	16.1	- 0.65
Group 3	4-43	9.3	- 8.50
Group 4	4-15	26.7	+ 14.08

First Time Out	W-R	%	£1 Level Stake
2-y-o	1-15	6.7	- 2.00
3-y-o	0-24	-	- 24.00
4-y-o+	1-12	8.3	- 4.00
Totals	2-51	3.9	- 30.00

0-10	Doncaster	0-6	Epsom 0-6
0-3	Haydock	0-3	Leicester 0-3
0-1	Nottingham	0-1	

THE PLOT THICKENS

We are now at the point in programming our interactive character game where we can start to add elements of the 'plot'. Here, we examine the final decisions that need to be made in our character-handler program and outline the tree structures involved.

So far in this series, we have given our characters the means to manipulate objects. What we need now is to add the remaining routines, which were outlined in the last instalment. It's at this point that the advantages of adopting a modular design for our program become apparent.

The flow-chart shown in the diagram represents the complete decision-making process carried out by the character handler. You can see that it incorporates both the object manipulation tree, which we have already seen in action, as well as modules for dealing with the plot, 'object awareness', and interaction with other characters.

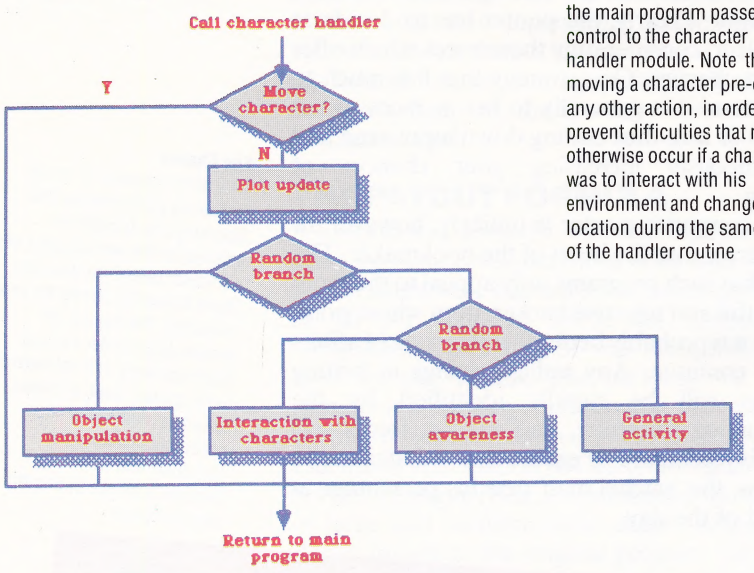
We have already discussed the rudiments of the plot — at some stage during the game, an unlucky character will eat a Dog and Bucket cornish pasty and die of food poisoning. The game will end when another character has gathered the necessary information to prove the guilt of Fred the Barman, whose habit of making pasties out of cat-food has brought about this unhappy turn of events. To solve the mystery, a character must find a cat food tin, see the victim, and put two and two together.

For programming purposes, the necessary information is stored in four main flags. We have already seen the first — numeric variable g. It's updated in line 5220 and temporarily stores the number of the character eating the cornish pasty for processing by the plot routines. Two other flags record the demise of a character and whether or not a character has spotted the victim. The fourth flag takes the form of an array DIMensioned to the number of characters with each element initialised to zero. When a character spots the victim, the relevant element is set to 255. We'll see all these at work in the full listing, which will be given in the next instalment.

The plot tree first checks to see whether the character has already been killed, and if so, the 'dead' flag will have been set to the number of the character. Since the program only allows for one victim, the plot routine will return at this point without taking further action

If, however, the 'dead' flag is not set, the routine continues to check whether or not the character has just taken a bite of the pasty. If the value of g matches the current character number, we then

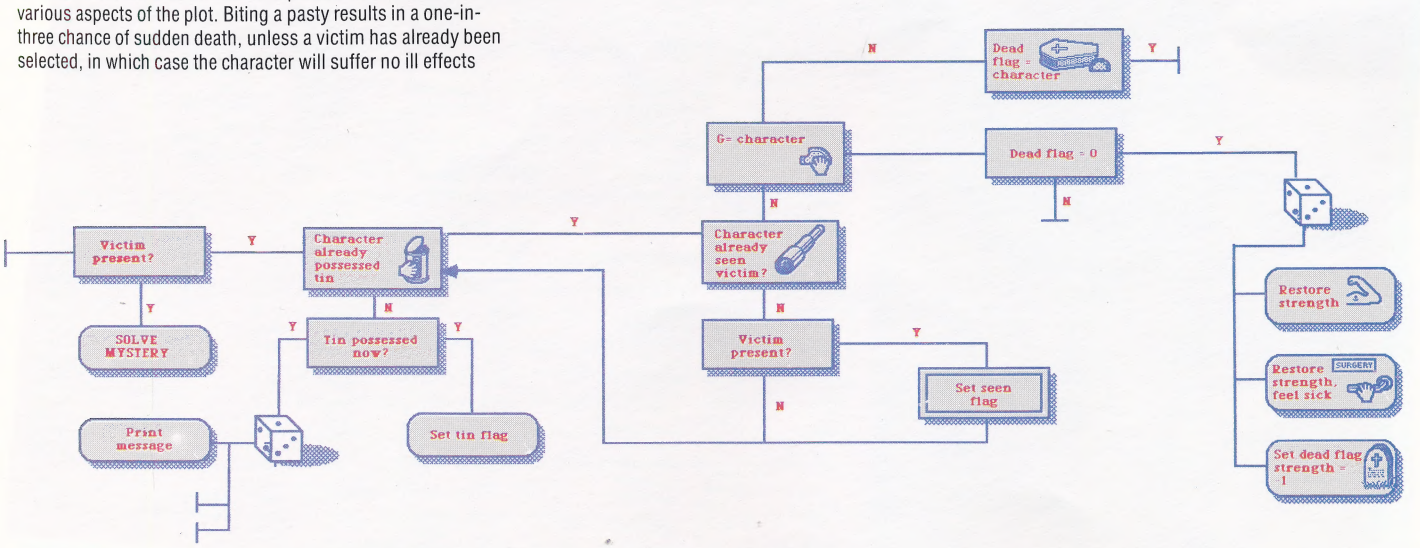
Handling The Cast



Our flow diagram shows the different actions taken when the main program passes control to the character handler module. Note that moving a character pre-empts any other action, in order to prevent difficulties that might otherwise occur if a character was to interact with his environment and change location during the same pass of the handler routine

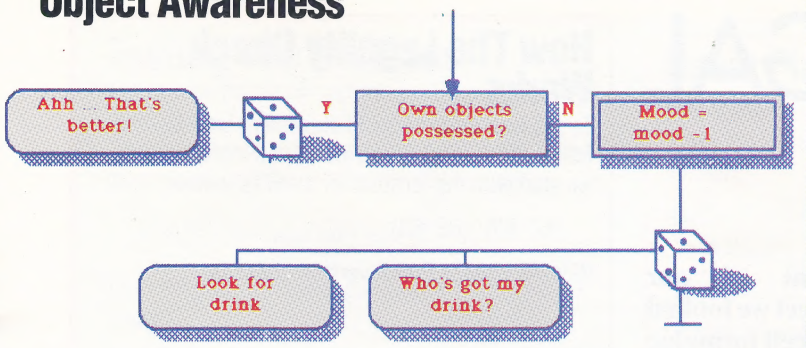
An Outline Of The Plot

We show here the flow of control required to deal with the various aspects of the plot. Biting a pasty results in a one-in-three chance of sudden death, unless a victim has already been selected, in which case the character will suffer no ill effects



ICONS BY IAN McKINNELL

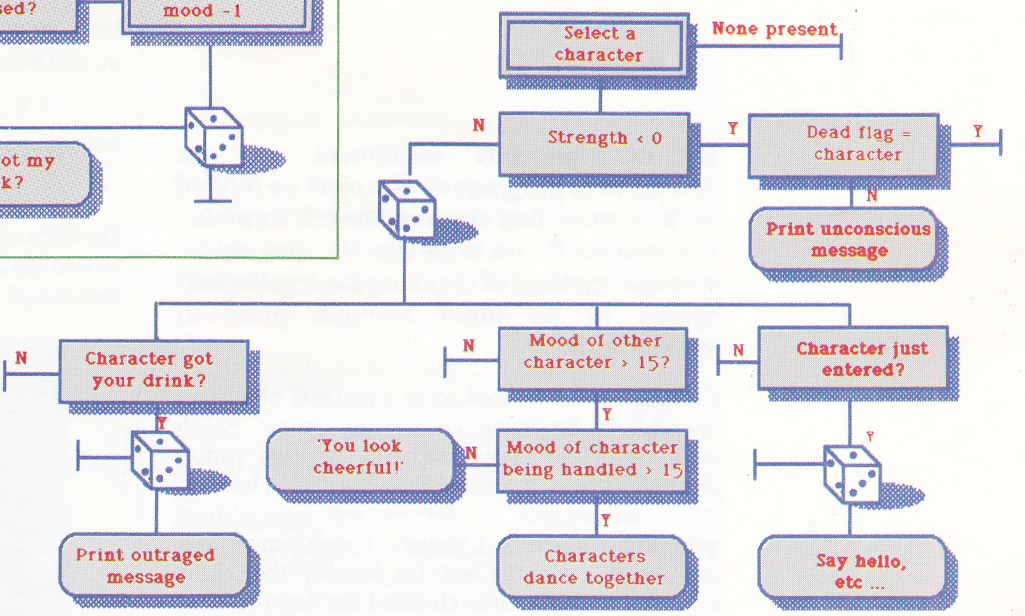
Object Awareness



Here we add some more object-oriented routines to those already provided by the Object Manipulation tree. In particular, the character's mood is decremented by one if that character does not have his own drink

Character Interaction

Our characters need to have some awareness of each other's moods and actions. This tree provides them with some opportunities to express themselves

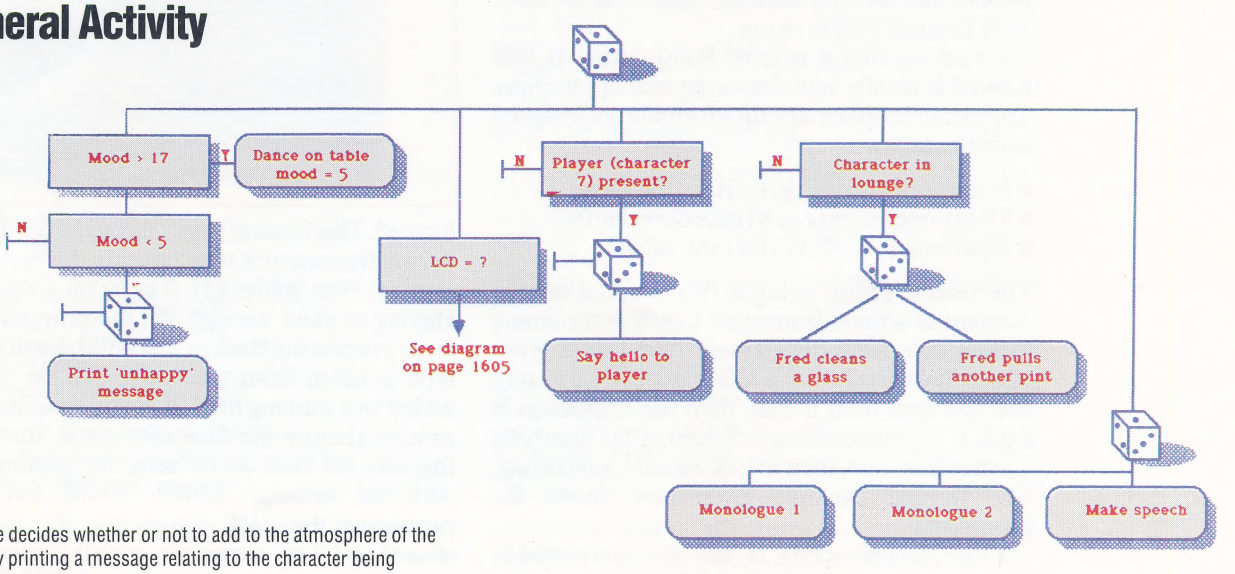


check to see whether a death has occurred (in which case the 'dead' flag will be greater than zero). If a death has not occurred, the program then branches at random to one of three conclusions. In the first two, the character will simply suffer no ill-effect or else have a sudden attack of stomach pain. In both these cases, the character's strength, which was diminished by 10 points in line 5230, will be restored to its previous level. In the third instance, the character's strength will be further reduced (if necessary) to -1 and the dead flag will be set to the character's number.

If the character is not eating the pasty, checks are then carried out to see whether or not other plot conditions have been fulfilled. You should be able to see exactly what happens by tracing the different paths through the tree. Note the use of a random node branching in three directions to give a one-in-three chance of printing a message, and the two terminal nodes on the bottom left-hand corner of the tree, which jump back into the tree to make further checks.

You should by now have no difficulty in 'reading' tree structure diagrams to see what processes are involved and what conditions are tested. Try to follow through the other trees shown and decide for yourself how they might be entered into our listing. You might like to consider the implications of traversing trees with different numbers of branches from different nodes.

General Activity



This tree decides whether or not to add to the atmosphere of the game by printing a message relating to the character being handled

CAROLINE CLAYTON

PERFECTLY LEGAL

In the previous instalment of our spreadsheet programming project we looked at the routine that converts the cell formulae into reverse Polish notation. We now devise a simple method of checking the legality and syntax of an input formula prior to evaluating it.

On page 1608 we looked at a method of writing speadsheet formulae known as reverse Polish notation. Converting 'normally' written (infix) formula strings to reverse Polish notation has two main advantages as far as our spreadsheet program is concerned. Reverse Polish expressions are much easier to test for legality than infix expressions and, once checked for legality, they are much easier to evaluate. We will look in detail at how values can be substituted into a formula and the way a formula is evaluated in a future instalment; here we show you how a reverse Polish string can be checked for legality.

Most BASIC programmers will have come across the error message SYNTAX ERROR on numerous occasions while typing in programs. The most common causes of syntax errors are typing errors (hitting the wrong key) and missing out vital information. When a spreadsheet user enters a formula into a cell, the same problems can occur. For example, a bracket may be missed off or there may not be enough operands for the operations to work on (try adding one number to a blank). A simple method exists for checking a reverse Polish string to ensure that the correct components are present and are in a sensible order — a so-called 'well formed' Polish string.

Checking that a reverse Polish string is well formed is ideally suited to computer application. The elements that make up a formula are assigned values as follows:

- Binary operators (e.g. +,–) take the value –1
- Unary operators (e.g. &) take the value 0
- Operands (e.g. A2, 27) take the value 1

The reverse Polish string is then worked on one element at a time, from right to left — a running total is kept from the element type values given above. If each subtotal is less than or equal to zero and the final total is one, then the expression is legal. If any subtotal exceeds zero or the final total is other than one, then the expression is incorrect. The worked example given here shows the procedure.

Once the infix string, I$, has been converted to its reverse Polish equivalent, P$, the subroutine at line 4700 can check to ensure that P$ is well formed. This routine works through P$ — from the *left* an element at a time, placing the appropriate element type values (–1, 0 or 1) on a stack, ST(). Having worked through P$, the program moves on to process the stack at line 4800. Each element type is taken from the stack, ST(), in turn and added to a running total. If at any time during this process (before the final element is 'unstacked') the subtotal rises above zero, the routine aborts with the message ERROR BEFORE END. After processing the stack completely, the final total should be 1 if the expression is well-formed. If this is not the case, an error message ERROR AT END is generated.

How The Legality Check Works

To see how the process of checking a well-formed Polish string works, let's look at a simple example. If we start with the formula for a cell as follows:

(A2+A3)*(B2–B3)

then the reverse Polish version of this is:

A2A3+B2B3–*

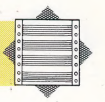
The following list shows how the reverse Polish version can be scanned from the right, keeping subtotals of element type values as it goes:

Element	Element Type Value	Subtotal
*	–1	–1
–	–1	–2
B3	1	–1
B2	1	0
+	–1	–1
A3	1	0
A2	1	1

When we scan a reverse Polish string from the right, we should always encounter an operator before the operands it relates to. Thus the subtotal should either be negative or, if both operands have been encountered, zero. The exception to this is the final total – which must be one — since there is always one more operand in a string than there are binary operators.

If for some reason the final * operator was missing in the above example then the method would soon detect the error:

Element	Element Type Value	Subtotal
–	–1	–1
B3	1	0
B2	1	1**ERROR**
+		
A3		
A2		



Testing The Routines

The reverse Polish conversion and legality checking routines can be tested at this stage by making a simple alteration to the program. Type in the listing given here and then change line 4010 to read:

```
4010 LET I$="your infix string"
```

Then enter the following commands in immediate mode:

```
GOSUB 3000:GOSUB 4000:PRINT P$
```

Sinclair Spectrum:

```
4700>REM ************************
4701 REM * CHECK WELL FORMED    *
4702 REM * REVERSE POLISH       *
4703 REM ************************
4705 GO SUB 4900
4710 LET P=0: LET SP=1: LET L1=1
: LET U$=""
4720 LET P=P+1: IF P>LEN (P$) TH
EN  GO TO 4800
4730 LET T$=P$(P)
4740 IF T$="+" OR T$="-" OR T$="
*" THEN  LET S(SP)=-1: LET G$(SP
)=T$: LET SP=SP+1: GO TO 4720
4745 IF T$="/" OR T$="^" THEN  L
ET S(SP)=-1: LET G$(SP)=T$: LET
SP=SP+1: GO TO 4720
4747 IF L1=LP THEN  GO TO 4720
4750 LET U$=E$(L1)
4751 FOR Z=1 TO LEN (U$)
4752 IF U$(Z)=" " THEN  GO TO 47
54
4753 NEXT Z
4754 LET U$=U$(1 TO Z-1)
4760 IF U$<>P$(P TO P-1+LEN (U$)
) THEN  LET U$="": GO TO 4720
4770 LET S(SP)=1: LET G$(SP)=U$
4775 LET SP=SP+1
4780 IF L1<LP THEN  LET L1=L1+1
4785 LET U$=""
4790 GO TO 4720
4800 REM *** PROCESS STACK *****
4810 LET P=S(SP)
4820 FOR C=SP-1 TO 1 STEP -1
4830 LET P=P+S(C)
4840 IF P>0 AND C<>1 THEN  PRINT
 AT 20,0;"ERROR BEFORE END": RET
URN
4850 NEXT C
4860 IF P<>1 THEN  PRINT "ERROR
AT END  ": RETURN
4870 LET CP=SP-1: RETURN
4900 REM *** MAKE LIST OF OPREAN
DS** ***
4905 LET P=0: LET LP=1: LET T$="
": LET U$=""
4910 LET P=P+1: IF P>LEN (I$) TH
EN  GO TO 4995
4920 LET T$=I$(P)
4930 IF T$="+" OR T$="-" OR T$="
*" OR T$="&" THEN  GO TO 4960
4940 IF T$="/" OR T$="^" OR T$="
(" OR T$=")" THEN  GO TO 4960
4950 LET U$=U$+T$: GO TO 4910
4960 IF U$="" THEN  GO TO 4910
4970 LET E$(LP)=U$: LET LP=LP+1
4980 LET U$=""
4990 GO TO 4910
4995 IF U$="" THEN  RETURN
4997 LET E$(LP)=U$: LET LP=LP+1
4998 RETURN
```

Basic Flavours

BBC Micro:

Make the following changes to the Commodore 64 version:

```
4840 If P>1 AND C<>1 THEN PRINT
   TAB(0,22);"ERROR BEFORE END
        ":RETURN
4860 IF P<>1 THEN PRINT TAB(0,22);"
   ERROR AT END        ":RETURN
```

Amstrad CPC 464/664:

Make the following changes to the Commodore 64 version:

```
4840 IF P>1 AND C<>1 THEN LOCATE
   1,22:PRINT"         ERROR BEFORE
   END        ": RETURN
4860 IF P<>1 THEN LOCATE 1,22:PRINT"
   ERROR AT END        ":RETURN
```

Commodore 64:

```
4700 REM ******************************
4701 REM **** CHECK FOR WELL FORMED ***
4702 REM *** REVERSE POLISH STRING ***
4703 REM ******************************
4705 GOSUB 4900
4710 P=0:SP=1:L1=1:TE$=""
4720 P=P+1:IF P>LEN(P$) THEN 4800
4730 T$=MID$(P$,P,1)
4740 IF T$="+" OR T$="-" OR T$="*" THEN
ST(SP)=-1:G$(SP)=T$:SP=SP+1:GOTO 4720
4745 IF T$="/" OR T$="^" THEN ST(SP)=-1:
G$(SP)=T$:SP=SP+1:GOTO 4720
4746 IF T$="&" THEN ST(SP)=0:G$(SP)=T$:S
P=SP+1:GOTO 4720
4747 IF L1=LP THEN 4720
4750 LET TE$=E$(L1)
4760 IF TE$<>MID$(P$,P,LEN(TE$)) THEN TE
$="":GOTO 4720
4770 LET ST(SP)=1:LET G$(SP)=TE$
4775 LET SP=SP+1
4780 LET L1=L1-(L1<LP):TE$=""
4790 GOTO 4720
4800 REM *** PROCESS STACK ***
4810 LET P=ST(SP)
4820 FOR C=SP-1 TO 1 STEP -1
4830 LET P=P+ST(C)
4840 IF P>0 AND C<>1 THEN GOSUB 1950:PRI
NT "    ERROR BEFORE END    ":RETURN
4850 NEXT C
4860 IF P<>1 THEN GOSUB 1950:PRINT "
  ERROR AT END        ":RETURN
4870 LET CP=SP-1:RETURN
4900 REM ** MAKE LIST OPERANDS IN I$ **
4905 LET P=0:LET LP=1:LET T$="":LET TE$=
""
4910 FOR K=1 TO 20:LET E$(K)="":NEXT K
4915 LET P=P+1:IF P>LEN(I$) THEN 4995
4920 LET T$=MID$(I$,P,1)
4930 IF T$="+" OR T$="-" OR T$="*" OR T$
="&" THEN 4960
4940 IF T$="/" OR T$="^" OR T$="(" OR T$
=")" THEN 4960
4950 LET TE$=TE$+T$:GOTO 4915
4960 IF TE$="" THEN 4915
4970 LET E$(LP)=TE$:LP=LP+1
4980 LET TE$=""
4990 GOTO 4915
4995 IF TE$="" THEN RETURN
4996 LET E$(LP)=TE$:LET LP=LP+1
4997 RETURN
```

As operands in the expression can take the form of cell names, such as A2 or L14, it is important to isolate them before checking for legality. Since it is much easier to identify such operands while the expression is in its infix form (as operands are all separated by operators), the subroutine at line 4900 works on the infix version of the formula, I$, storing the operand names it finds in E$().

SLAVE

In large computer systems or networks, the various computers are often configured so that each unit within the system is handling a different area of processing. When there is a central computer or processor performing the tasks of designating processing or handling communications between the various computers and peripherals, what's known as a 'master-slave' system is in use. In this case, the central computer or processor is known as the 'master' and the other machines are referred to as the *slaves*. Less important tasks can be delegated to the slave systems while the job of collating all the results is reserved for the master.

Of course, developing such a system brings problems of its own, primarily one of synchronisation. Because the master machine can only deal with one task at a time, the slave systems will often be idle while awaiting further instructions.

SOFT KEYBOARD

The characters generated on a *soft keyboard* can be altered by the software, in contrast to the keyboards used on electronic typewriters, for instance, whose keys are 'hard-wired' to the characters. Soft keyboards are far more adaptable than the hard-wired variety since not only the original layout of the keyboard can be altered for any character set, but the keyboard can be adapted for various applications packages in which 'single keypress functions' might be advantageous.

SOFTWARE

The term *software* was originally coined to distinguish the electronic signals that constitute the program within the computer from the 'hardware' — the physical components of the computer. Since then, the word has, like so much computer jargon, lent itself to an entire range of other terms. For example, a company that specialises in writing computer programs is known as a 'software house', while computer programmers (especially those involved in mainframe system software maintenance) are known as 'software engineers'. Similarly, the programs they use, such as assemblers, monitors and debuggers, are known as the 'software engineering environment'.

Computers use two types of software. 'Systems software' are the programs used to run the computer, which consist of the operating system and other programs that enable you to communicate with the computer, including the Basic Input Output System (BIOS) and the high-level language compilers and interpreters. This term used to refer to all the programs within the computer. These days, however, many such programs used on microcomputers are held in ROM, and have been dubbed the 'firmware'.

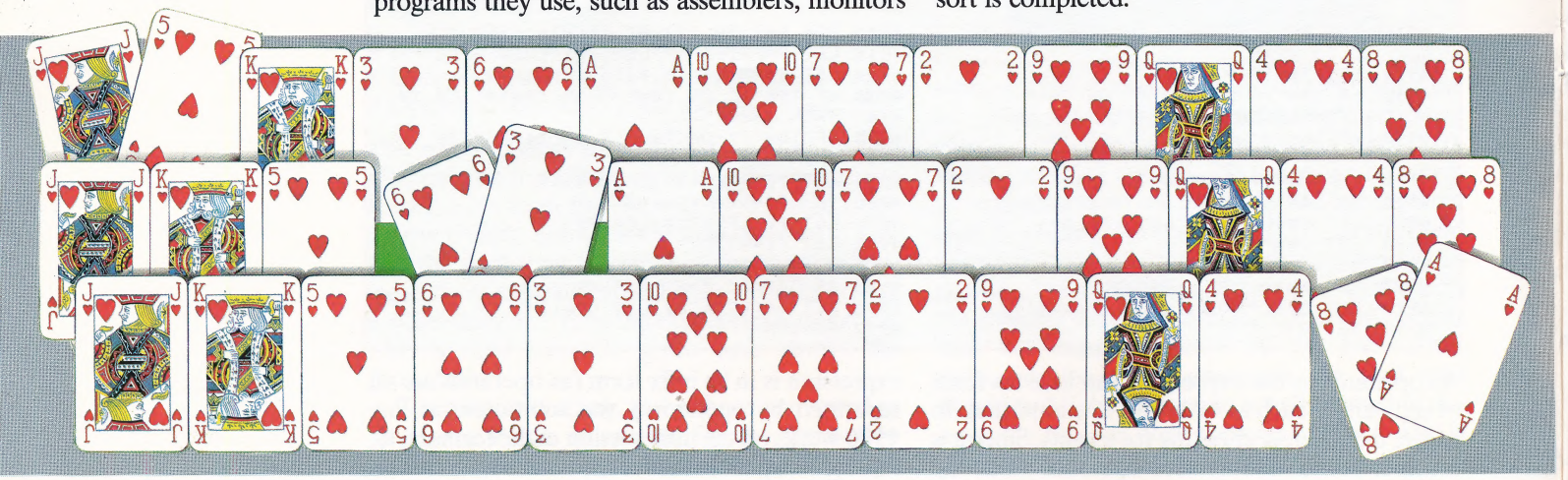
The other type of program is known as 'applications software'. These are programs that are designed to achieve a particular purpose, whether it is word processing, database management or constructing an impenetrable defense against space invaders.

SORTING

Sorting is the process of arranging a number of items into an ascending or descending order according to a chosen system. In more complex database systems, the items will be sorted according to the 'sort key' that has been selected. The sort key, usually alphabetic or numeric, is a part of the item that's common to all the other items, although its value will be different for each one. Thus the sort keys for all the items have to be of the same type so that an analysis can be made as to whether the sort key on one item is greater than, less than or equal to another.

A wide range of sorting algorithms have been devised over the years. These range from simple insertion methods of sorting to sophisticated tree selection sorts. However, all sorting systems use the same basic format. The computer will hold a list of the items in its memory and will reserve a register or other memory location for a single item. An item from the list will be selected, according to the method being used, and the sort key copied into the register or memory location. A comparison will then be made with the sort keys of the other members of the list and a value will be assigned to the item. The process will then be repeated with another member of the list until the sort is completed.

Bubbling Under
A bubble sort can be used to sort a complete suit of cards into order so that the king is on the left and the ace is on the right. The sort works through the cards from the left end comparing each pair in turn and swopping them if they are in the wrong order. During the first pass the ace will be repeatedly moved to the right eventually 'bubbling' its way to the right-hand end. Repeated passes must be made until all the cards are in order



BANK MANAGER

While the Amstrad CPC 464 and 664 have had excellent success, they have nevertheless lacked what many in the industry believe to be a major component of an all-round micro — enough memory to be a practical small business machine. The new CPC 6128 is hoped by Amstrad to find a place in this larger market.

A year after launching the CPC 464 (see page 429) in June 1984, Amstrad had captured 25 per cent of the home computer market in the UK. The success of this cassette-based machine was based on the same concept as Amstrad's hi-fi products — boxing all the system components together, providing the facilities most users want and selling the unit at a competitive price. Amstrad's next micro was an enhanced version of the CPC 464, which included an integral disk drive and slightly expanded BASIC. The CPC 664 (see page 1490) and the DDI-1 add-on disk drive for the 464 (see page 1209) featured CP/M and a version of Digital Research's Dr LOGO. But only a few months later, Amstrad launched the CPC 6128, a similar machine in most respects to the 664, but with 128 Kbytes of RAM. The 6128 computer has a more serious look about it. Gone are the coloured control keys, which are replaced by a smart uniform grey keyboard. On the CPC 464/664, a separate numeric keypad and cursor key cluster were provided. These keys have been integrated on the 6128 into a single bank, making the overall width of the new machine about two and a half inches less than the 464. The height of the case has also been reduced so that the keys now rest at a more agreeable typing height. Furthermore, the keys have less travel, providing a much more 'secure' feel when typing.

As with the CPC 664, second disk drive, expansion and Centronics printer ports are provided at the rear of the machine, but the cassette, joystick and audio ports have been moved to the left-hand side of the case, while the power and volume controls have moved from the right-hand side to the rear. The disk drive is the single-sided 3in used by the CPC 664, but the mechanism has been housed in a much thinner case and topped with number reference charts for the keys and screen colours.

Two disks are supplied with the CPC 6128. The first of these includes on one side an enhanced version of CP/M, called CP/M Plus. This is backed with a number of programming utilities including an assembler, disassembler and disk housekeeping utilities in addition to the usual CP/M utilities, such as PIP and SUBMIT. The second disk has a 48 Kbyte version Dr LOGO, which is a considerable improvement over the version packaged with the CPC 664 and DDI-1, with over 50 additional commands and primitives.

CHRIS STEVENS

New Arrival
Amstrad's CPC 6128 computer builds on the success of the CPC 464 and 664 machines, maintaining software compatability with its predecessors while increasing its memory to 128 Kbytes. The enhanced version of CP/M bundled with the 6128 allows it to run classic CP/M business packages

Also included on the top side of the second disk is Digital Research's GSX program and a HELP facility. GSX was the precursor to DR's GEM environment, although it didn't prove very popular with software houses. The idea behind GSX is that it provides a 'device transparent graphics interface', which means that graphics routines written using GSX on one Z80- or 8086-based machine will run under GSX on any other similarly based machine.

Although the addressing capacity of an eight-bit processor is 64 Kbytes, the 6128, based on the eight-bit Z80A, somehow manages double this amount. This is made possible by the process known as bank switching (see page 429), in which different areas of ROM and RAM can be switched 'in' and 'out' of the 64 Kbyte addressing space of the processor.

The BANKMAN utility from the systems disk adds extra commands to BASIC enabling it to handle the extra memory. The 128 Kbytes of RAM are arranged into two 64 Kbyte sections, but BASIC

CP/M Plus Much More

CP/M Plus, the new version of CP/M provided with the CPC 6128 has several advantages over CP/M 2.2 used with the CPC 664 and 464 with DDI-1 disk drive. Most importantly, CP/M Plus utilises the extra memory on the 6128 to free 61.5 Kbytes of RAM for applications programs once it's installed. This is three to four Kbytes more than what's required by standard CP/M programs and so makes available wealth of CP/M business software, including SuperCalc, dBase II and WordStar.

Many CP/M packages are written for twin or double-sided disk drives and attempting to access a drive that was not present resulted in an error. CP/M Plus has been modified to get around this problem by displaying corrective action messages so that the appropriate disk can be inserted manually if required.

Other features of CP/M Plus include a facility to install foreign language character sets and another to act as a VT52 terminal to a mini or mainframe computer

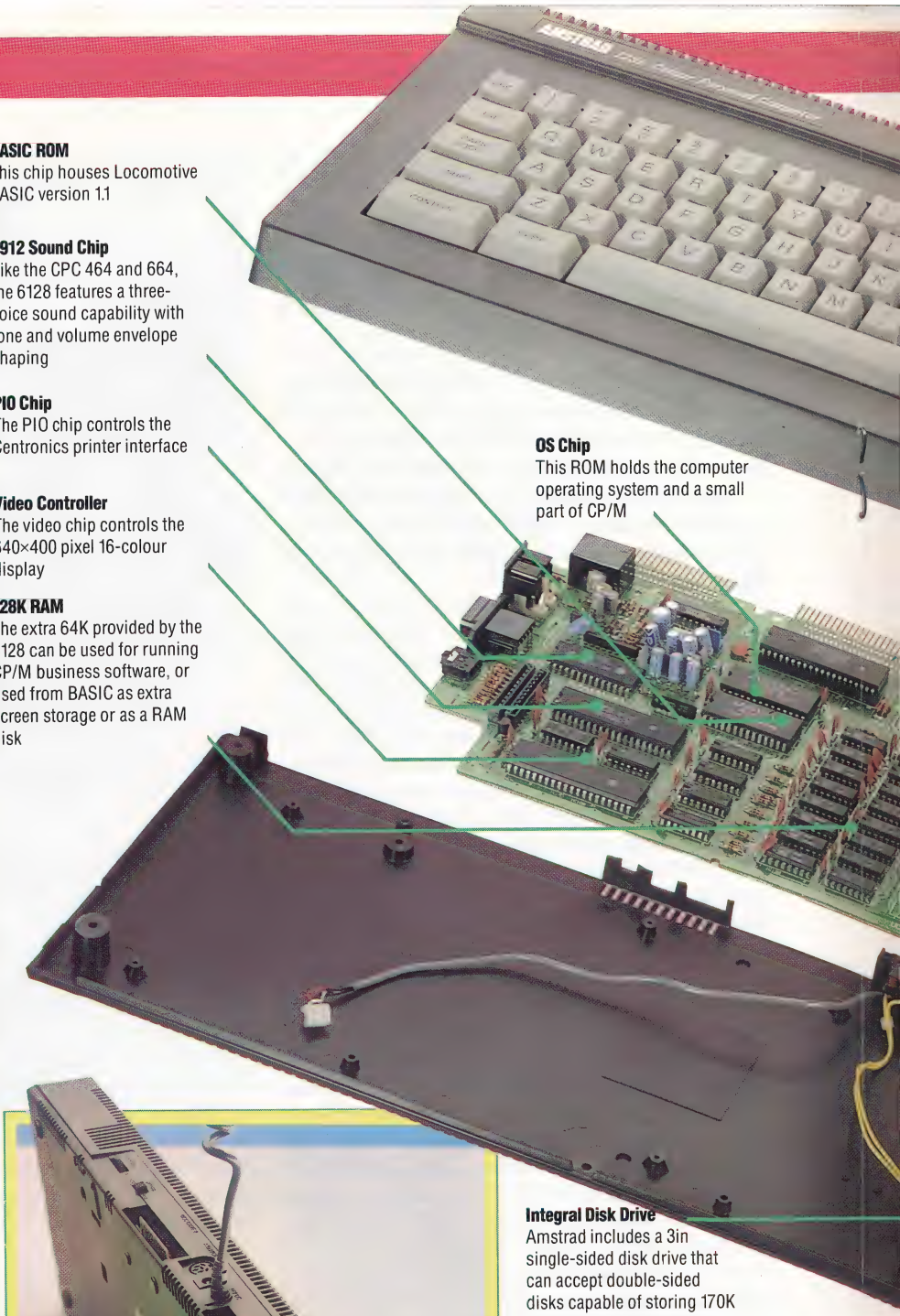
Room To Work

The enhanced version of Dr LOGO supplied with the CPC 6128 includes many commands left out of the original implementation on the Amstrad CPC 664. Extra list processing, arithmetic, graphics, editing and disk commands are incorporated to make it a much fuller and more useful version of this increasingly important language. The overall size of the workspace has increased from 2,105 nodes on the CPC 664 to 3,753.

Particularly welcome are facilities that allow access to a printer and I/O ports, the latter allowing floor turtles to be easily controlled from LOGO. Improved disk-handling commands are included that let files on a disk be renamed and alternative drives selected from within the language. Improved editing and debugging commands allow files to be loaded straight from disk into LOGO's screen editor and global variables and procedures to be listed within the workspace.

programs can only occupy one of these sections, preventing you from writing bigger BASIC programs on the 6128 than you could on the CPC 664 or 464. However, the 64 Kbyte section that isn't used for programs can be used as a storage area that can be accessed from BASIC. The new commands allow this extra memory to be used either to store extra screen displays, which can be switched into the normal screen area, or to act as a record filing system.

Because the screen display requires 16 Kbytes, the extra 64 Kbytes section is able to hold four extra screens, numbered 2 to 5. Screen 1, the standard screen, is held in the other 64 Kbyte section used by BASIC. The command SCREENCOPY,A,B copies screen B into screen A's 16 Kbyte area, overwriting the original contents, while SCREENSWAP,A,B exchanges the contents of the



BASIC ROM
This chip houses Locomotive BASIC version 1.1

8912 Sound Chip
Like the CPC 464 and 664, the 6128 features a three-voice sound capability with tone and volume envelope shaping

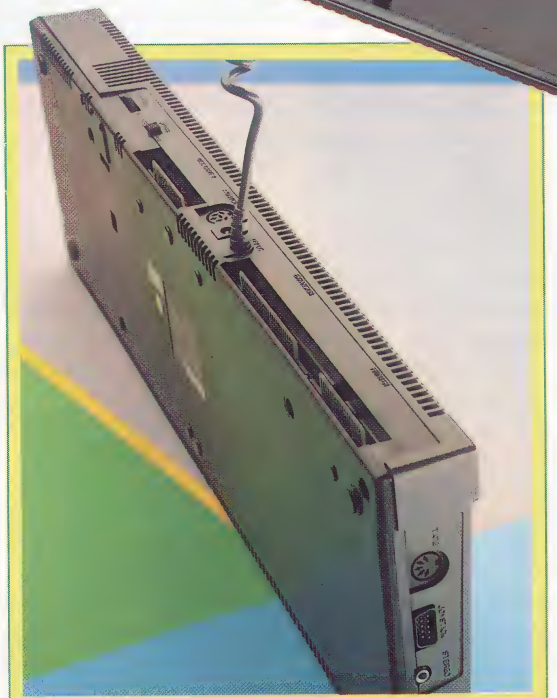
PIO Chip
The PIO chip controls the Centronics printer interface

Video Controller
The video chip controls the 640×400 pixel 16-colour display

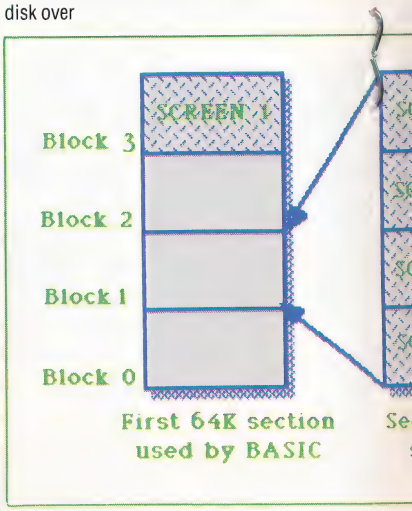
128K RAM
The extra 64K provided by the 6128 can be used for running CP/M business software, or used from BASIC as extra screen storage or as a RAM disk

OS Chip
This ROM holds the computer operating system and a small part of CP/M

Integral Disk Drive
Amstrad includes a 3in single-sided disk drive that can accept double-sided disks capable of storing 170K on each side, although the underside of the disk can only be accessed by turning the disk over



Port Holes
The 6128 features Centronics printer, expansion, second disk drive, joystick, stereo sound and cassette ports. 5v, 12v and monitor connections are also present and attach to the monitor supplied



two screen areas. A screen-swap can take around half a second, but it's possible to swap or copy one sixty-fourth of the screen at a time. It's likely that games writers will take full advantage of these facilities for rapid changes in small portions of the screen.

The extra 64 Kbyte section can also be used as a file, in which case the extra memory is arranged as a number of records where BASIC strings can be stored — the RAM disk, so called because of its similarity to the structure of a disk file. Records within the file must all be the same length, which is set up by the IBANKOPEN, n statement, where n is the length of each record between two and 255 characters. IBANKREAD and IBANKWRITE allow string data to be passed to or from the RAM disk and an optional record number can be passed with these commands that specifies the record to be used. If no record number is specified, BASIC will start writing or reading using the last record specified (or the first record) and automatically increment a pointer ready to deal with the next record. In other words, either a random access or a sequential file can be set up and used.

The CPC 6128 is a well-styled, compact machine that represents great value for money. The main criticism of the CPC 664 was that although it was provided with CP/M, it did not have sufficient memory to be considered as a serious computer for small business use. The expansion to 128 Kbytes of RAM, together with an enhanced version of CP/M that's capable of utilising the extra memory, enables the 6128 to run many standard CP/M packages. The new machine is also software compatible with the CPC 464 and 664 and therefore has a number of tailor-made software packages available for it. Based on these attributes, the CPC 6128 is an extremely attractive proposition for the user who wants an all-round business and games machine.

AMSTRAD CPC 6128

PRICE

With colour monitor: £399; with green monitor: £299. Both inc VAT

MEMORY

128K RAM, 48K ROM. Only 64K is available for BASIC programs but the other 64K can be used as a RAM disk or for storing four alternative screen displays

CPU

Z80A running at 4MHz

DISK

One single-sided 3in drive with a port for a second drive

INTERFACES

Expansion bus, second floppy disk interface, cassette, stereo, joystick and Centronics printer ports, 12v and 5v inputs, monitor socket

SOFTWARE PROVIDED

Two disks containing CP/M Plus, programming utilities, enhanced Dr LOGO and GSX. CP/M 2.2 and slimmed down Dr LOGO are also provided for compatability with the 664 and 464

DOCUMENTATION

The user instruction manual covers programming in BASIC and the memory management extensions to the system. Sections on Dr LOGO and CP/M cover the basics of operation but more detailed manuals on these subjects are available from Amstrad

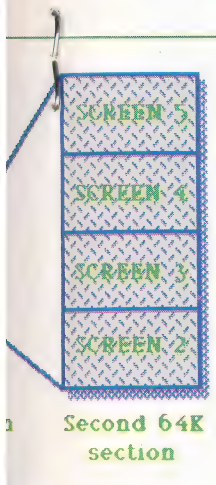
STRENGTHS

The CPC 6128 represents the limit of 8-bit computer technology and is extremely good value. Its ability to run standard CP/M packages together with its excellent BASIC and graphics make it equally useful for business and games applications

WEAKNESSES

Amstrad has again used 3in drives rather than the more popular 3½in drives used by many small business machines. Because the drives are single-sided and some programs use more than one face of a disk, users are likely to be engaged in much disk handling

Change Of Address



The BASIC supplied with the 6128 cannot directly access all 128K of RAM and can only work on programs within the first 64K section. The other 64K can, however, be used as a RAM disk or to store four alternate screen displays. As the Z80A processor can only address 64K at a time, extra memory is divided into 16K blocks so that any block can be temporarily switched into the Z80's address space between &4000 and &7FFF

A MUSICAL ADAPTATION

The MIDI interface board we designed and built earlier in the course (beginning on page 1335) was intended for the Commodore 64 and BBC Micro. We outline here the changes required for adapting the board to the Amstrad CPC range of micros, beginning with the construction and testing stages.

The interface hardware design is fundamentally the same for the Amstrad as it is for the BBC Micro and Commodore 64 version. The principal differences arise from the fact that the Amstrad machine is based on the Z80 processor, rather than on the 65XX used in the target machines for the original design. The Z80 is therefore not bus-compatible with the MC6850 ACIA device used in the interface (see page 1343).

The bus interface circuit requires an extra component: an IC containing a trio of three-input NAND gates. The two ACIA chip select lines, CS0 and CS1, are also required (in the original circuit they were connected directly to +5v). The ACIA chip enable line is basically the inverse of the Z80 $\overline{\text{IORQ}}$ (I/O request) line, but it is gated by M1 to prevent I/O access during the Z80 interrupt acknowledge sequence during which $\overline{\text{IORQ}}$ and $\overline{\text{M1}}$ simultaneously go low. It is also gated by address A7.

The Amstrad expansion port does not provide a suitable decode line for direct connection to external I/O devices and so the decoding must be done externally. I/O addressing on the CPC range utilises all 16 address bits (the contents of register B are output on the upper eight address bits during most Z80 I/O instructions). Addresses available for use by the expansion bus are from &F800 to &FBFF while the lower eight bits must be between &E0 and &FE for user peripherals.

This isn't as complicated as it may seem, because all internal I/O addresses have address bit A10 at +5v. Therefore, to decode our address range, we need to detect a low level on A10, and high levels on A5 to A7. This is done by connecting A10 to $\overline{\text{CS2}}$, and A5 and A6 to CS0 and CS1 respectively, while address A7 is used to 'gate' the enable (E) signal, preventing it from becoming active unless A7 goes high.

Unlike the 6502 processor, the Z80 does not provide a single read/write signal, so our design treats the read and write registers of the 6850 as different addresses by tying the R/$\overline{\text{W}}$ line to address line A9. The internal register select line is connected to A8, which results in the ACIA registers being allocated I/O addresses as shown in the ACIA Register Allocation table.

It's important to note that because the action of reading or writing to the ACIA is controlled by an address line, it isn't possible to write to the addresses of read only registers (&FAE0, &FBE0). This will result in both the Z80 and the ACIA trying simultaneously to place a byte of data on the system bus. It's even more dangerous to try to read from the write-only addresses (&F8E0, &F9E0). This will result in a write being performed to the ACIA with the data bus in an unpredictable floating state.

The board should be tested in the same way as for the earlier version. The procedure is presented again here with the appropriate adjustments for the Amstrad.

1. Connect a standard 5-pin DIN cable between the IN and OUT sockets on the board.
2. Initialise the ACIA with the following command:

```
OUT &F8E0,3
```

3. Set up the clock rate, serial word format and

Parts List

The following parts can obtained from Maplin:

No	Description	Maplin Number
1	1nF polycarbonate capacitor	YY24B
2	100nF polycarbonate capacitors	YY11M
1	270 ohm resistor	M270R
3	220 ohm resistors	M220R
2	680 ohm resistors	M680R
2	PCB mounting 180° 5-pin DIN sockets	YX91Y

The following parts can be obtained from Technomatic, 17 Burnley Road, London NW10 1ED:

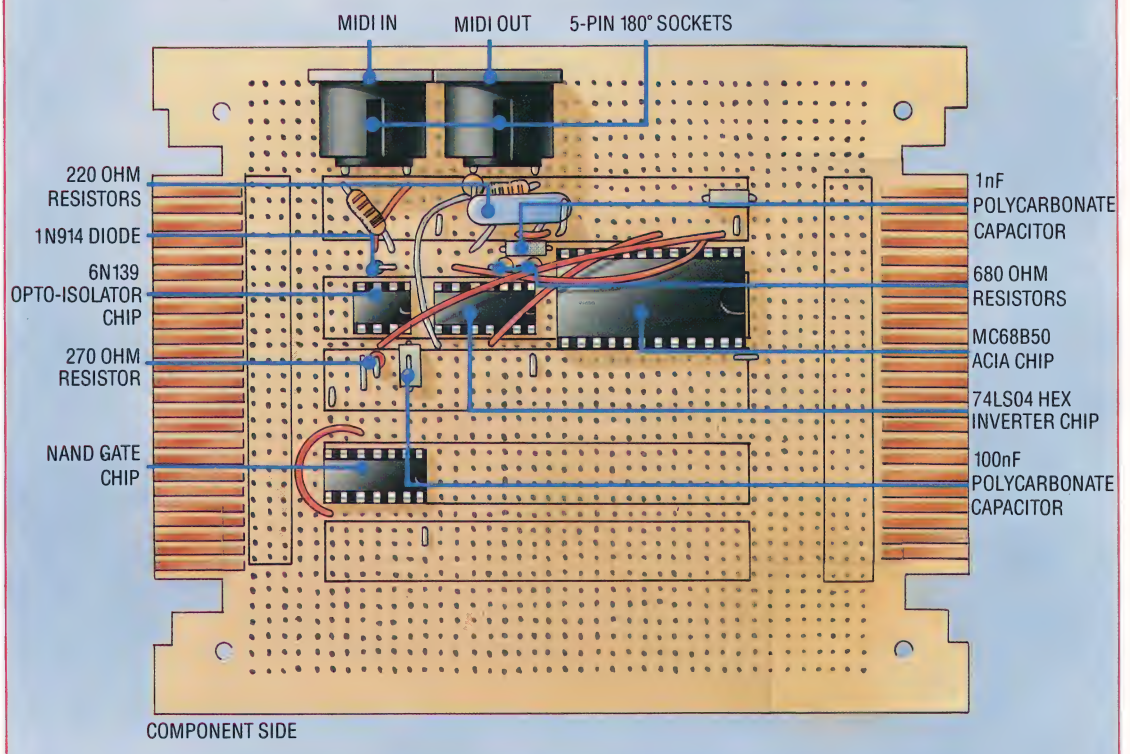
1 MC68B50 ACIA chip
1 6N139 opto-isolator chip
1 74LS04 TTL hex inverter
1 74LS10N triple three-input NAND gate
1 24-pin DIL socket
2 14-pin DIL sockets
1 8-pin DIL socket
1 2.0 MHz crystal
1 1N914 diode

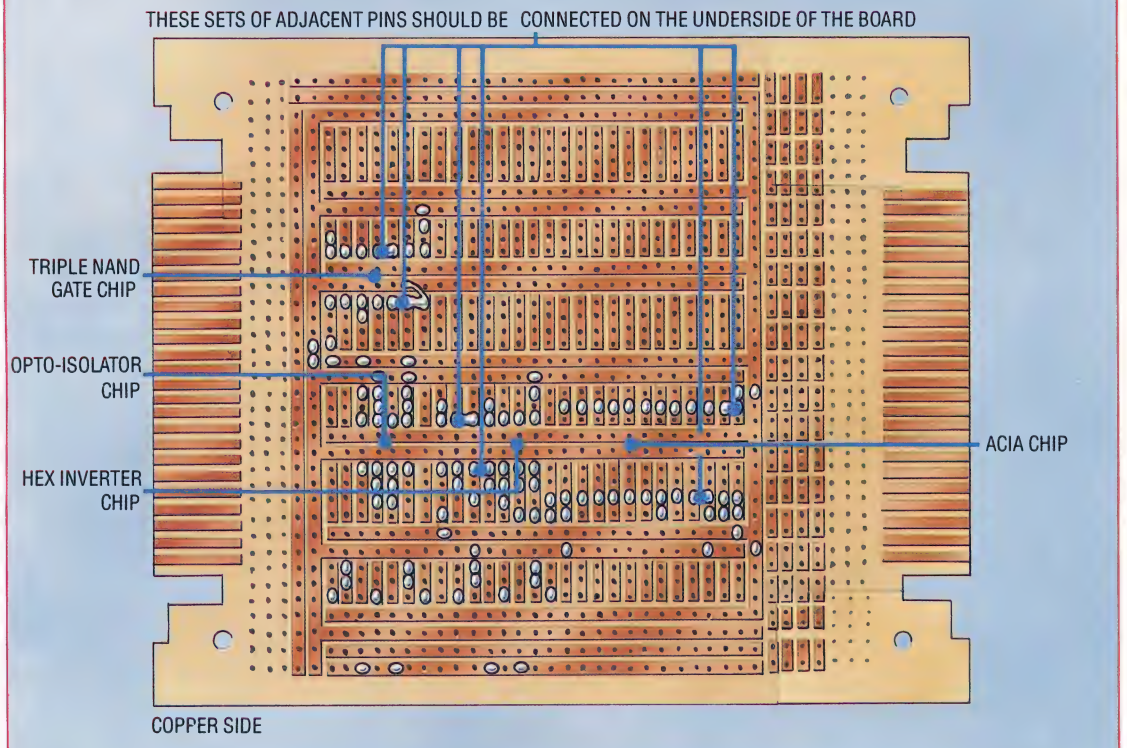
The DIP breadboard used to mount the components can be obtained from most branches of Tandy: part number 276-164.

The connecting cable components can be obtained from many sources:

2 50-way IDC edge connectors
30cm 50-way ribbon cable

Component Layout Diagram





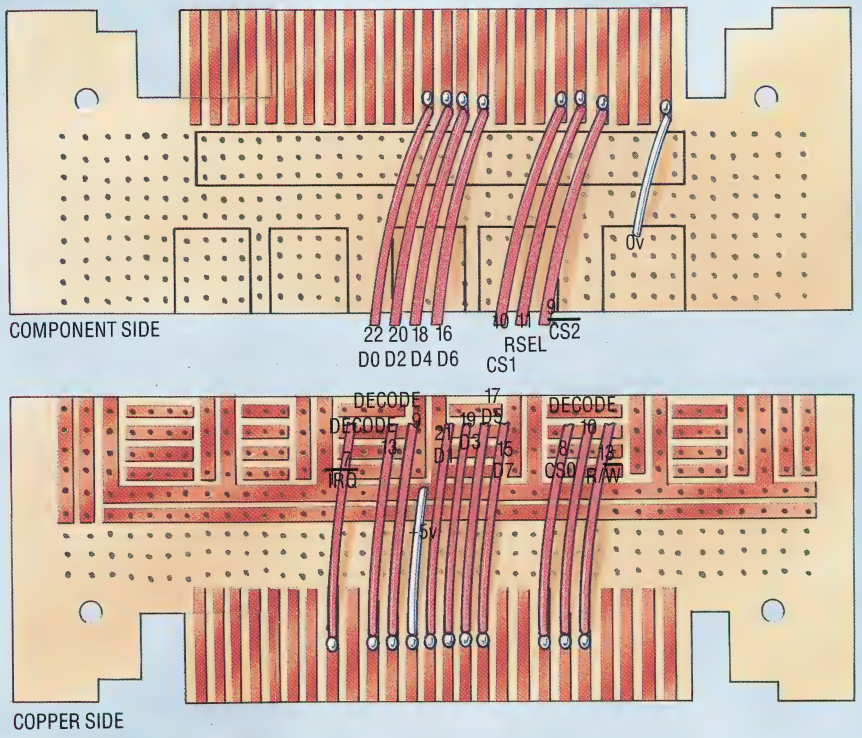
KEVIN JONES

Construction Details

The major components of the MIDI interface should be mounted on the special DIP breadboard. Start by mounting the passive components: the resistors, capacitors, DIL and DIN sockets. Make the necessary on-board links using covered wire-wrap wire and mount the crystal. Ensure that the diode is mounted so that the end marked with a coloured band is uppermost and that you do not miss the small link under the capacitor nearest the hex inverter chip

Connecting Edge
The left-hand end of the board is used to accept signals from the Amstrad expansion bus. The diagrams here show the appropriate bus connections to the ACIA chip, 74LS10 decoder chip and the power feeds. When all these links are made with covered wire-wrap wire, the four ICs can be gently pushed into their sockets, taking care to note the correct orientation of the notches. Finally, make up a 50-way connecting cable using two 50-way IDC edge connectors and a 30cm length of ribbon cable. Push one connector onto the board and the other into the expansion port of your Amstrad, ensuring that the computer is off. The board is now complete and ready for testing

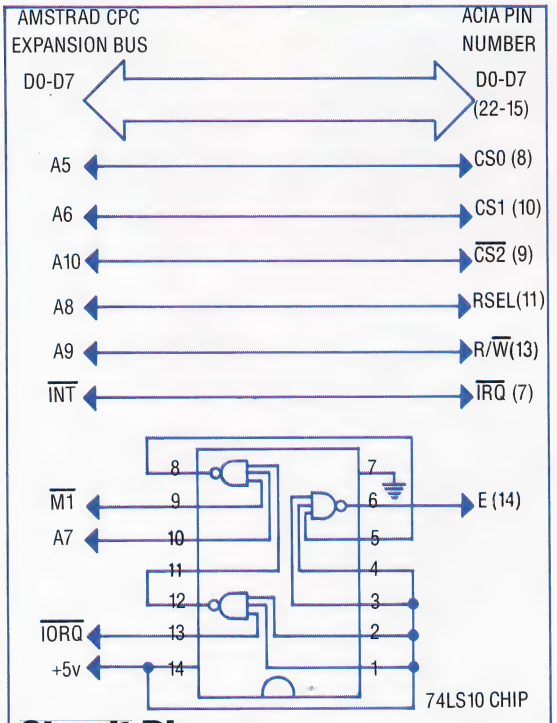
Edge Connections



ACIA Register Allocation

Address	Register	
&F8E0	Control	(WRITE ONLY)
&F9E0	Transmit data	(WRITE ONLY)
&FAE0	Status	(READ ONLY)

Amstrad Decoding
Our original MIDI interface circuit design (see page 1343) was designed for 65XX-based machines. To use the interface with a Z80-based machine, such as one of the Amstrad CPC range, the computer bus signals have to be decoded prior to connection with the ACIA chip. The circuit diagram here shows the necessary decoding logic which is achieved in our amended design using a single triple three-input NAND gate chip



KEVIN JONES

Circuit Diagram

disable receive/transmit interrupts by typing:

```
OUT &F8E0,&16
```

4. Read the ACIA status register with:

```
PRINT INP(&FAE0)
```

The correct value should be two, which indicates that the bus connection is functioning correctly.
5. Send a serial byte from the output to input by typing:

```
OUT &F9E0,x
```

where x can be any integer between zero and 255.
6. Correct reception of the byte is verified by reading the status register again:

```
PRINT INP(&FAE0)
```

This should return the value three.
7. Also verify that the byte is the expected value by reading the receive data register:

```
PRINT INP(&FBE0)
```

This should return the same value x as used in step 5. Steps 4 to 7 may be repeated as required with various values of x.

If any of these tests fail, or if the computer hangs up with the board connected, check your wiring with a multimeter if one is available. If problems still persist, refer to page 1368 on which some of the likely problems were discussed.

In the final instalment of this project we will create software for our Amstrad MIDI interface, including a program that takes advantage of the operating system's interrupt structure.

THE FORTRAN DIMENSION

While the only data structure in FORTRAN is the array, the language maintains its power through a complex use of subroutines. We look at some of the available statements as well as the various methods employed by FORTRAN that make it such a suitable language for numerical applications.

The only data structure available to the FORTRAN programmer is the array, and as with BASIC, it must be declared before it is used — since FORTRAN is a compiled language this means right at the beginning of the program. This is done using a DIMENSION statement, which works in the same way as a BASIC DIM statement.

Using the default data types, in which any variable name beginning with I, J, K, L, M or N is integer and anything else is real, a statement such as:

```
DIMENSION A(20), I(50)
```

will reserve memory space for an array of up to 20 real numbers and another array of up to 50 integers. If you want to override the default typings with INTEGER, REAL, DOUBLE PRECISION, COMPLEX or LOGICAL statements, then one of these must be given in addition to the DIMENSION statement. It's possible, however, to make the declaration of the array while making the declaration of type:

```
INTEGER ARR1(20)
REAL NUMS(30)
```

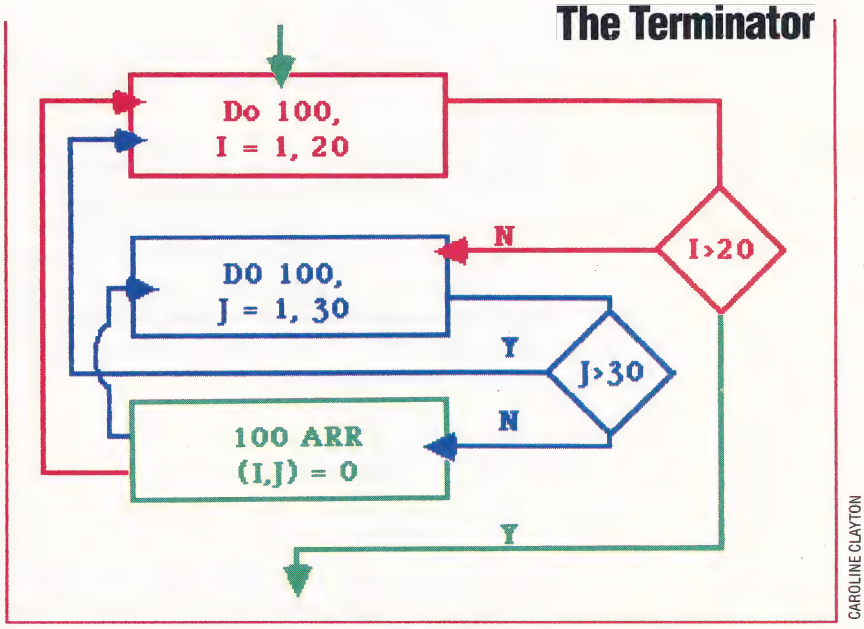
This reserves space for an array of 20 integers and one for 30 real numbers.

Two-dimensional arrays are possible and some versions allow for three or even more dimensions. Two-dimensional arrays are declared in the normal way. Here are two examples:

```
DIMENSION ARR(20,30)
INTEGER ARR(20,30)
```

Any integer constant or variable can be used to index the array, such as ARR(3,4) or INTARR(I).

Although there's no such thing as a character string in FORTRAN IV, this deficiency was remedied among other things in FORTRAN 77, which we'll look at more closely in the next instalment. Characters are stored as integers in ordinary integer variables; we saw in the last instalment (see page 1613) how one or more characters could be stored in any integer variable. The closest that FORTRAN comes to a character string is an array of integers, but this is not really satisfactory for anyone accustomed to the comprehensive string-handling facilities offered by nearly all BASICS.



CAROLINE CLAYTON

The main looping facility and the only control structure in FORTRAN IV (apart from the IF and GOTO statements) is the DO loop, which effectively operates in the same fashion as a BASIC FOR. . .NEXT loop. The DO loop takes the form of:

```
      DO Sn (intvar) = startval, finishval, stepval
      .....
Sn    (last statement in block to be repeated)
```

Sn is the number of the last statement in the block that is to be repeated. Any integer variable can be used as a loop counter (intvar) and the starting, finishing and step values are specified. Our DO loop would appear in BASIC as:

```
FOR intvar = startval TO finishval STEP stepval
.....
NEXT intvar
```

To make the value of an array zero, for example, we would enter:

```
      DO 100 I = 1,20,1
100   ARR(I)=0
```

where the step value is one then it can be omitted as in:

```
      DO 100 I = 1,20
100   ARR(I) = 0
```

The statement that's numbered to mark the end of the loop may be any executable statement, except for an IF, GOTO or another DO.

The fact that there is no clear end-of-loop statement is sometimes useful if you want to

Compact Code

FORTRAN has no clear end-of-loop statement, and more than one loop can share the same terminating statement. This can lead to some very compact coding, as shown in this example of program flow where two loops share the same terminator. The BASIC equivalent, though less compact, is far easier to read and hence to debug:

```
100 FOR I=1 TO 20
110 FOR J=1 TO 30
120 LET A(I,J) =0
130 NEXT J
140 NEXT I
```

produce the most compact code, but it can also lead to some horrendous errors as well as code that is absolutely incomprehensible even to the author, particularly when two or more loops are nested inside each other. For example, this initialisation of a two-dimensional array is perfectly legal:

```
      DO 100 I =1,20
      DO 100 J=1,30
100   ARR(I,J)=0
```

(Note that the same statement has been used as the terminator in both loops.)

To overcome the restrictions on the final statement in a DO loop and to provide clearer code, the 'dummy' CONTINUE statement is often used as a terminator. It is a legal executable FORTRAN statement that does absolutely nothing and may be used anywhere in a program as needed. Normal convention requires us to terminate DO loops with their own unique CONTINUE, so the previous example would become:

```
      DO 101 I=1,20
        DO 100 J=1,30
          ARR(I,J)=0
100     CONTINUE
101   CONTINUE
```

Some programmers go further and insist that all transfers of control, such as DOs, IFs and GOTOs, should terminate on a CONTINUE, which certainly helps to make FORTRAN programs more understandable.

As a mathematical language, you would expect FORTRAN to provide some additional features for the handling of arrays of numbers, particularly in input/output. The whole of a one- or two-dimensional array can be read or written in a single READ or WRITE statement, where the associated FORMAT gives the layout of each line or record. For example:

```
      DIMENSION IARR1(20)
      ..........
      WRITE(1,10)IARR1
```

would cause the entire contents of IARR1 (20 integer values) to be output. Furthermore:

```
10 FORMAT (20I4)
```

would produce 20 four-digit integers across one line;

```
10 FORMAT (I4)
```

would give you one per line;

```
10 FORMAT (5I4)
```

would give you four lines of five numbers; and:

```
10 FORMAT (20A1)
```

would give you a string of 20 characters on one line.

There's one problem when two-dimensional arrays are input or output using this method — namely, FORTRAN, unlike nearly all other languages, stores two-dimensional arrays in column-first order rather than row-first. If you're not careful, you'll find yourself working with a transposed version of what you really want. To cater for this problem, and for situations in which you're not dealing with the whole of an array, a special form of DO loop is provided, called the 'implied DO loop', which is written with the variable name in the READ or WRITE statement. For example, to fill up the array IARR2(10,20) with 10 strings of 20 characters could be done with:

```
      DO 100 I=1,10
      READ (1,10) (IARR2(I,J), J=1,20)
10    FORMAT (20A1)
100   CONTINUE
```

FORTRAN has a genuine, if somewhat restricted, subroutine facility. Subroutines are completely independent, normally using only local variables, and they can usually be appended to the end of a main program or written and compiled separately. It's quite in order to use the same line numbers again in a subroutine, and in most cases the same variable names, without the risk of confusion. Parameter passing is by reference only — in other words, the address of the parameter is passed to the subroutine when it is called so that any changes in value caused by the subroutine will affect the corresponding variable in the calling program. Global variables, which can be referenced in the subroutine and main program (see page 688), can be declared, if required, by means of a COMMON statement in both program modules. A number of COMMON blocks can be set up and named, but for most purposes a single unnamed block is sufficient.

Variables named in the COMMON statements in the two modules must agree in total usage of memory but don't have to agree in type, though this is not recommended as a type-changing mechanism. For instance, a main program could contain:

```
      COMMON I1,I2,I3
```

where the subroutine could contain:

```
      COMMON I(3)
```

The usage of memory is the same in both cases, but in the first, the three integer variables are kept distinct, whereas in the second they are referred to as elements of an array.

A typical subroutine call would take the form:

```
      CALL MYSUB(X,Y,I,J)
```

where the subroutine would begin with a statement:

```
      SUBROUTINE MYSUB(A,B,L,M,)
```

(Note that the parameters inside the brackets must agree in number and type.)

Control returns from the subroutine to the calling program by means of a RETURN statement (there may, in fact, be more than one RETURN in a subroutine.) One of these should be executed regardless of the control path that's followed

through the subroutine. The final statement in a subroutine subprogram, as in any main program, must be an END, though this is more of a compiler directive than an executable statement.

One interesting feature of parameter passing is that it becomes possible, in a sense, to change the size of an array dynamically at run-time. This is easily done in an interpreted language like BASIC, but it is very rare in a compiled language, where storage space must be allocated at compile time rather than at run-time.

As well as subroutines, FORTRAN provides a FUNCTION facility for subprograms that returns a single value, which is similar in many respects to that in PASCAL. A function subprogram is written separately, in a similar manner to a subroutine, but beginning with a FUNCTION statement. An assignment should be made to the function name before a RETURN is executed. The default type of value returned by the function is determined by the function name using the usual convention. This requirement may be overridden if necessary, though it is not advisable unless a function used by a variety of different programs requires it. Functions may be used in the main program in the same way as library functions (simply by using their name, with a parameter list) in any expression where it is valid to use a variable of that type.

We'll conclude our look at FORTRAN in the next instalment by looking at its file-handling facilities and consider the extra features added in the 1977 standard. We'll also have a look at FORTRAN compilers on microcomputers.

Finding The Roots

```
C   A FORTRAN PROGRAM TO READ SETS OF
C   THREE VALUES A,B,C, REPRESENTING
C   THE COEFFICIENTS OF A QUADRATIC
C   EQUATION A*X**2+B*X+C=0 USING THE
C   USUAL FORMULA
    FUNCTION DISCR(A,B,C)
    DISC=B*B-4.0*A*C
    RETURN
    END
    SUBROUTINE SOLVE(A,B,C,X1,X2,NROOT)
    D=DISCR(A,B,C)
C   NOTE THAT AN ARITHMETIC IF COULD BE
C   USED HERE BUT IT IS NOT RECOMMENDED
    IF (D.GE.0.0)GOTO 100
C   D IS LESS THEN ZERO SO COMPLEX ROOTS
    NROOT=0
    X1=0.0-B/(2.0*A)
    X2=SQR(ABS(D))
    GOTO 300
100 IF (D.GT.0.0)GOTO 200
C   D IS ZERO SO EQUAL ROOTS
    NROOT=1
    X1=0.0-B/(2.0*A)
    X2=X1
    GOTO 300
C   D IS POSITIVE SO TWO ROOTS
    NROOT=2
200 X1=(0.0-B+SQRT(D))/2.0*A
    X2=(0.0-B-SQRT(D))/2.0*A
300 RETURN
    END
C   FIRST READ NUMBER OF SETS OF VALUES
    READ(2,10)NVAL
C   PRINT HEADINGS
    WRITE(3,11)
    DO 500 I=1,NVAL
C   READ THREE VALUES AND CHECK IF
C   ACCEPTABLE
100     READ (1,12) A,B,C
        IF (A.EQ.0.0) GOTO 500
C   VALUES ARE SUITABLE SO CALL SUBROUTINE
        CALL SOLVE(A,B,C,X1,X2,NROOT)
        IF (NROOT.GT.0)GOTO 200
C   COMPLEX ROOTS
        WRITE (1,13)A,B,C,X1,X2
        GOTO 500
200 IF (NROOT.GT.1)GOTO 300
C   EQUAL ROOTS
        WRITE(1,14)A,B,C,X1,X2
        GOTO 500
C   TWO ROOTS
300 WRITE(1,15)A,B,C,X1,X2
        GOTO 500
C   INVALID VALUES FOR A,B AND C
400     WRITE (1,16)A,B,C
500 CONTINUE
    STOP
C   FORMAT STATEMENTS
10  FORMAT (I4)
11  FORMAT(1H ,8X,1HA,8X,1HB,8X,1HC,8X,
    4HTYPE,8X,2HX1,8X,2HX2)
12  FORMAT(1H ,3F7.2)
13  FORMAT(1H ,4X,3(F7.2,2X),4X, 7HCOMPLEX,
    X,F7.2,3X,F7.2)
14  FORMAT(1H ,4X,3,(F7.2,2X),4X,5HEQUAL3X,
    F7.2,3X,F7.2)
15  FORMAT(1H ,4X,3,(F7.2,2X),4X,
    7HUNEQUAL,X,
    F7.2,3X,F7.2)
16  FORMAT(1H ,4X,3,(F7.2,2X),4X,7HINVALID)
    END
```

Average Calculation

```
C   A FORTRAN FUNCTION TO CALCULATE THE
C   (REAL) AVERAGE OF A SET OF N INTEGER
C   NUMBERS IN AN INTEGER ARRAY IARR
    FUNCTION AVGE(IARR,N)
    DIMENSION IARR(N)
    SUM=0
    DO 100 I=1,N
      SUM=SUM+FLOAT(IARR(I))
100 CONTINUE
    AVGE=SUM/FLOAT(N)
    RETURN
    END
```

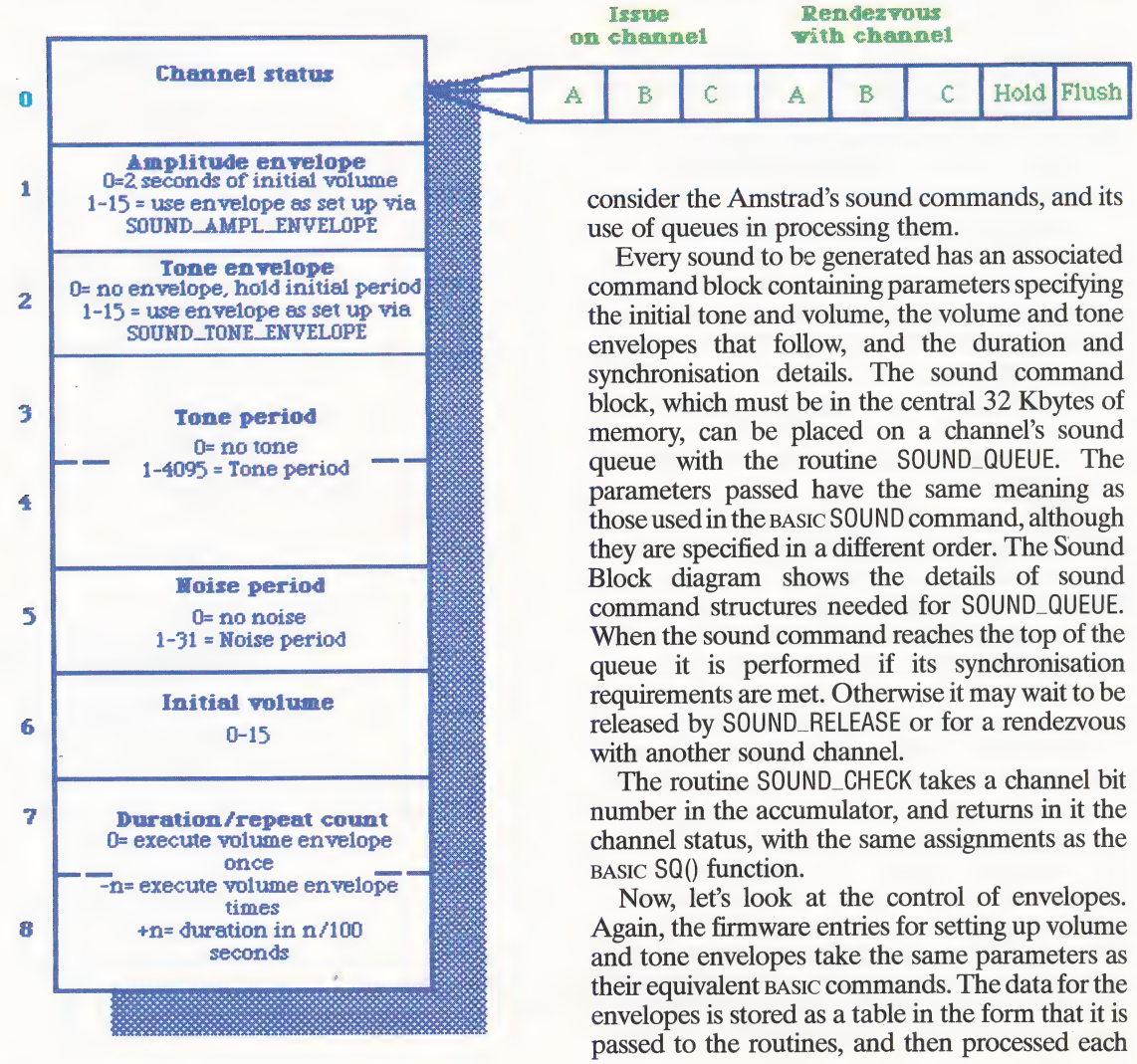
SOUND SYSTEM

The operating system used by the Amstrad CPC range of computers provides sound facilities that have parallels with BASIC sound commands. Here, we look at the way the OS controls sound, and show you how to use it to generate continuous background music.

The hardware design of the Armstrad range of CPC computers uses the popular General Instruments AY-3-8912 programmable sound generator (known as the '8912'), which is used not only to generate sound, but additionally to scan the keyboard using the eight-bit port on the chip.

The 8912 supports three independent sound channels (A, B, C) and one pseudo-random noise generator, which may be programmed to appear on any channel. The chip has the basic facilities to produce a tone and to change the volume of the tone, using predefined volume envelopes. The outputs from the sound generator are mixed to give a stereo output: channel A is left, channel B is right, while channel C is mixed equally between the two. These facilities have been extended and improved upon by the operating system, which provides software control of tone and volume together with methods of synchronising the sound channels.

The BASIC commands ENV, ENT, RELEASE, SOUND, and SQ make direct use of the operating system to support sound. In fact the parameters passed and the concepts involved are almost identical. The operating system deals with sound control by way of a special interrupt event that occurs one hundred times a second. This event is dedicated to process sound commands — for example, channels are rendezvoused and envelopes updated during this event. First, let's consider the Amstrad's sound commands, and its use of queues in processing them.



Sound Comand Block
A sound command consists of a block of data containing various parameters, identical to those used in the BASIC SOUND command. The address of this block is passed in HL to SOUND_QUEUE, which then takes appropriate action. If the queue is full the command is ignored and the carry flag returns false

Every sound to be generated has an associated command block containing parameters specifying the initial tone and volume, the volume and tone envelopes that follow, and the duration and synchronisation details. The sound command block, which must be in the central 32 Kbytes of memory, can be placed on a channel's sound queue with the routine SOUND_QUEUE. The parameters passed have the same meaning as those used in the BASIC SOUND command, although they are specified in a different order. The Sound Block diagram shows the details of sound command structures needed for SOUND_QUEUE. When the sound command reaches the top of the queue it is performed if its synchronisation requirements are met. Otherwise it may wait to be released by SOUND_RELEASE or for a rendezvous with another sound channel.

The routine SOUND_CHECK takes a channel bit number in the accumulator, and returns in it the channel status, with the same assignments as the BASIC SQ() function.

Now, let's look at the control of envelopes. Again, the firmware entries for setting up volume and tone envelopes take the same parameters as their equivalent BASIC commands. The data for the envelopes is stored as a table in the form that it is passed to the routines, and then processed each

time the envelope is required. The address of the data block associated with an envelope can be determined at any time with SOUND_T_ADDRESS or SOUND_A_ADDRESS and so it is therefore possible to alter an envelope without recalling the firmware by patching the data block directly.

The data block format is shown in the Envelope Data Format diagram; any section can be patched by simply calculating the required offset in the data block. Care must be taken if this method is adopted as it is possible that the data area could be simultaneously accessed by the firmware while it is being patched. Although no disastrous effects will occur, the sound being produced may well not turn out as expected.

SOUND EVENTS

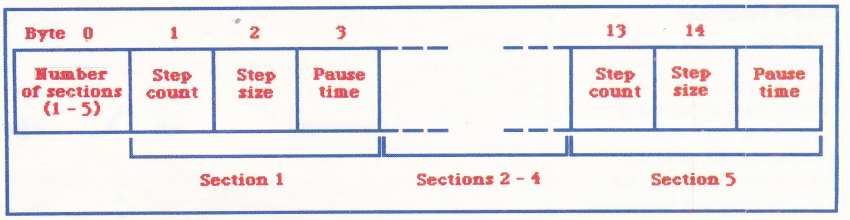
A particularly useful routine provided in the firmware is the SOUND_ARM_EVENT routine. This sets up an event to be kicked when the sound queue for a specified channel next becomes empty. In fact, the routine logs a special event onto the sound processing interrupt, which occurs every one-hundredth of a second. This event checks the sound queue of the relevant channel and if there is space then the event block passed to the SOUND_ARM_EVENT routine is kicked. This kicked event is then processed in a way specified by its event class.

The event block passed must be initialised first using KL_INIT_EVENT and so it may have any class or priority. However, because the sound interrupt occurs every one-hundredth of a second it is advisable to set the event as a normal asynchronous type.

This event can be used to generate continuous background music to a program without the programmer having to worry about continually providing the sound manager with data. Instead, the event routine should read the next sound program address from a pre-defined data area and then call the SOUND_QUEUE entry. The routine can be arranged to cycle round to the start of the data when it reaches the end to provide the continuous music effect.

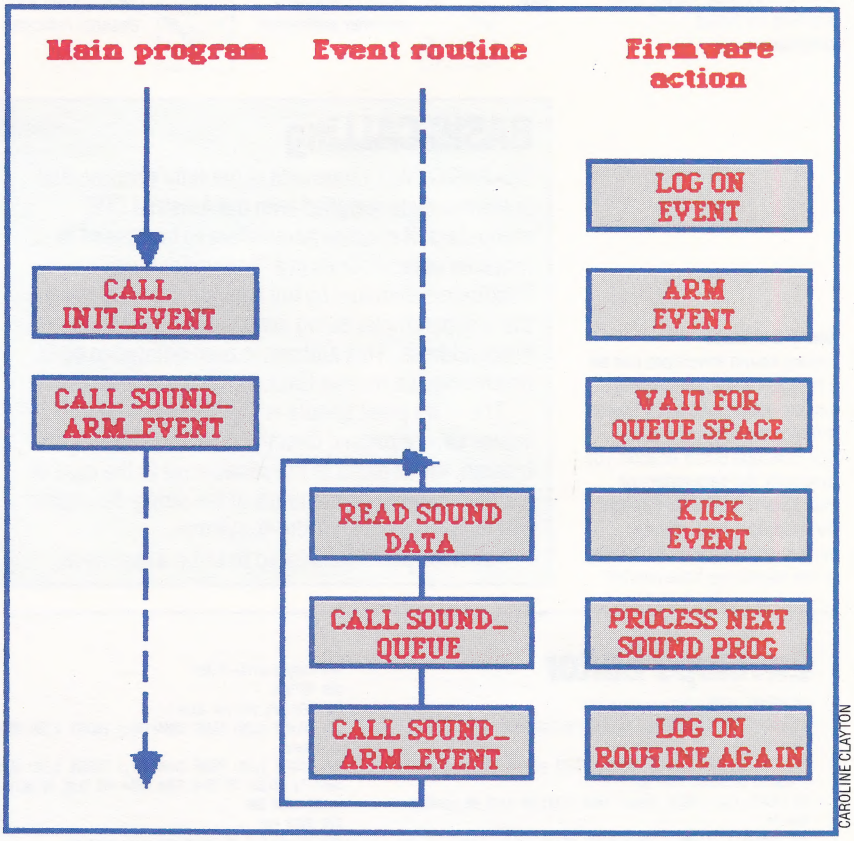
The SOUND_QUEUE entry disables the sound event, so before the routine finishes it must log itself on again by calling SOUND_ARM_EVENT passing its own event block address. The flow diagram shows how the event routine interacts with the sound queue routine to generate a continuous music effect.

Sound generation can be halted at any time using the SOUND_HOLD entry; this suspends any envelopes that are currently in use and disables the sound event (if armed). The sound can then be restarted by calling SOUND_CONTINUE, SOUND_QUEUE or SOUND_RELEASE. The last of these routines is provided to start any sound programs that are flagged to be held individually. SOUND_HOLD is called whenever a break is detected from the keyboard, to ensure that spurious sound generation from within a program is not allowed to continue.



▲ Envelope Data Block
Each envelope can have up to five sections, each of which is specified by three bytes determining step count, step size and pause time. If hardware volume envelopes are used, the first byte of the block holds the envelope number (between 128 and 143) and bytes two and three hold the pause time

▼ Notable Event
Using SOUND_ARM_EVENT enables the programmer to generate event-driven music on the Amstrad CPC computers to run in the background while the main program continues to execute in the foreground. The diagram shows the different operations performed by the main program, event routine and firmware

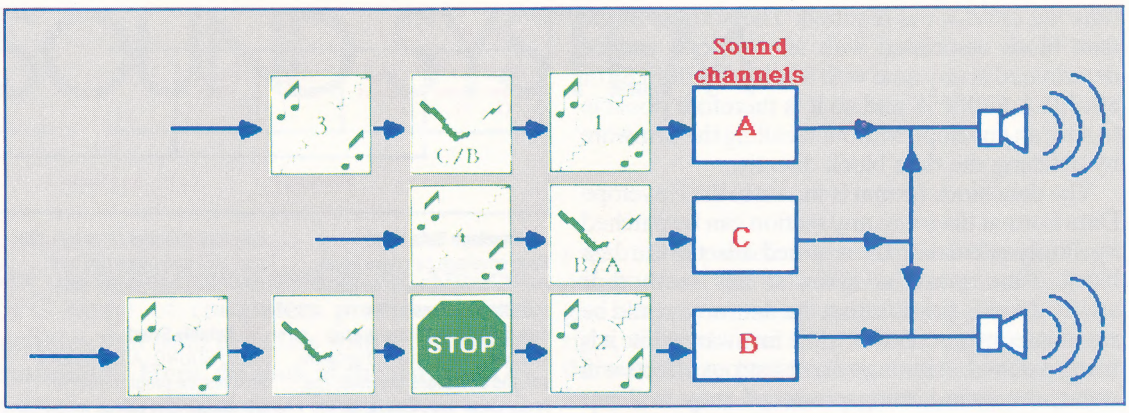


Sound Manager Entries

0BCA7H	SOUND_RESET	Resets the sound manager
0BCAAH	SOUND_QUEUE	Adds a sound to one of the sound queues
0BCADH	SOUND_CHECK	Returns the status of a sound queue
0BCB0H	SOUND_ARM_EVENT	Allows an event to be kicked when a sound queue is empty
0BCB3H	SOUND_RELEASE	Releases sounds held on a queue
0BCB6H	SOUND_HOLD	Stops all sounds on all queues
0BCB9H	SOUND_CONTINUE	Continues stopped sounds
0BCBCH	SOUND_AMPL_ENVELOPE	Initialises one of the software volume envelopes
0BCBFH	SOUND_TONE_ENVELOPE	Initialises one of the software tone envelopes
0BCC2H	SOUND_A_ADDRESS	Returns the address of a volume envelope
0BCC5H	SOUND_T_ADDRESS	Returns the address of a tone envelope

Full details of the entry and exit addresses of these routines are held in the Amsoft publication SOFT 158, the Amstrad Firmware Specification Manual

Channel Rendezvous
In the situation represented here, sounds One and Two are being executed. When Two finishes, the next command in the queue for channel B is held awaiting SOUND_RELEASE. Because the command in the queue for C requires rendezvous with channels A and B, it is held awaiting the last commands on those queues. Meanwhile, One finishes sounding and channel A will rendezvous with C, but still awaits its rendezvous with B. As soon as the queue for B is released the rendezvous is then true, and sounds Three, Four and Five are heard simultaneously



Sound And Vision
Editing sound envelopes can be a complex affair, particularly if you are unsure of exactly how to produce the effect you want. Our Envelope Editor enables you to specify the parameters of your choice, and then displays the envelope shape on the screen, showing you the 'shape' of the sound you have created

BASIC CALLing

The BASIC CALL command is not fully documented in the manuals supplied with the Amstrad CPC computers. It enables parameters to be passed to machine code routines in a 'parameter block'. Parameters supplied by the user are stored in order, the first parameter being held at the lowest (first) block address. This address is then pointed to by IX on entry to the routine CALLed.

The main point to note is that whereas numeric values can be passed directly, variables cannot. Instead, the address of the variable (or in the case of a string variable, the address of the string descriptor block) is passed using the @ symbol.

For example, if we wished to CALL a routine at &A000, passing to it the value 85, and return to BASIC with a result stored in the variables result% and answer$, we would enter the following:

CALL &A000,85,@result%,@answer$

On entry to the routine, IX would point to a block in RAM containing first the value 85 stored as a 16-bit unsigned integer, then two bytes containing the address of result%, and finally the address of the string variable descriptor block for answer$. The string descriptor block comprises three bytes holding first a value for the length of the string, and secondly its address. User routines should avoid altering the contents of this block and the processing of strings should therefore be carried out with care to avoid firmware problems.

Envelope Editor

```
50 MEMORY &6FFF: DIM envelope(15)
60 done=0: routine=&7000: buffer=routine+&100: nk=1
70 WHILE -1
80  MODE 2: LOCATE 1,20: INPUT "Start volume (0-15) ";sv%: I
F sv%)15 OR sv%<0 THEN 80
90 LOCATE 1,21: INPUT "Start tone ";tt: IF tt<0 OR tt)4095 T
HEN 90
100 LOCATE 1,22:INPUT "Horizontal scale (1-10) ";sc%: IF sc%
)10 OR sc%<1 THEN 100
110 LOCATE 1,23:INPUT "Volume or tone envelope (V/T) ";senv$
: IF (UPPER$(senv$)<>"V") AND (UPPER$(senv$)<>"T") THEN GOTO
 110
120 VV=0: IF UPPER$(senv$)="V" THEN pk=&BC: VV=1:  ELSE pk=&
BF
130  length%=0: soundx=0: IF VV=1 THEN soundy=INT(sv%*9.7) E
LSE soundy=68
140  FOR x=1 TO 15
150   envelope(x)=0
160  NEXT
170  GOSUB 270
180  LOCATE 1,20: PRINT CHR$(18)
190  FOR count=1 TO 5
200   GOSUB 360: nk=ABS(nk-1): GOSUB 540: IF VV=1 THEN GOSUB
 880 ELSE GOSUB 890
210  NEXT
220  WHILE NOT done
230   LOCATE 5,20: INPUT "Edit (y/n) ";ed$
240   IF UPPER$(ed$)="N" THEN done=-1 ELSE GOSUB 790
250  WEND
260 WEND
270 REM intialise screen
280 MODE 1: WINDOW #1,1,39,1,2: GOSUB 1000
290 MOVE 20,380: DRAW 20,210,3: DRAW 600,210,3
300 MOVE 21,soundy+211
310 POKE routine, &3E: POKE routine+1,1
320 POKE routine+2, &21: POKE routine+3, buffer-(INT(buffer/
256))*256: POKE routine+4, INT(buffer/256)
330 POKE routine+5,&C3: POKE routine+6,pk: POKE routine+7,&B
C
340 POKE routine+8,&C9
350 RETURN
360 REM get section data
370 LOCATE 1,20: PRINT CHR$(18);: LOCATE 5,20: PRINT "Sectio
n ";count
380 LOCATE 1,22: PRINT CHR$(18);: LOCATE 5,22: INPUT "Step c
ount "; skip%: IF VV=0 THEN GOTO 400 ELSE IF skip%<0 OR skip
%)127 THEN 380
390 GOTO 410
400 IF skip%<0 OR skip%)239 THEN GOTO 380
410 P$=STR$(SKIP%): GOSUB 900
420 LOCATE 1,23: PRINT CHR$(18);: LOCATE 5,23: INPUT "Step s
ize ";size%: IF VV=0 THEN GOTO 440 ELSE IF ABS(size%)>15 THE
N 420
430 GOTO 450
440 IF ABS(size%)>127 THEN 420
450 P$=STR$(size%): GOSUB 900
460 ss=size%: IF size%<0 THEN size%=256-ABS(size%)
470 LOCATE 1,24: PRINT CHR$(18);: LOCATE 5,24: INPUT "Pause
time "; pause%: IF pause%)255 THEN 470
480 P$=STR$(pause%): GOSUB 900
490 envelope ((count-1)*3+1)=skip%: envelope ((count-1)*3+2)
=size%: envelope ((count-1)*3+3)=pause%
500 FOR x=20 TO 24
510  LOCATE 1,x: PRINT CHR$(18);
520 NEXT
530 RETURN
540 REM envelope intialisation
550 POKE buffer,count
560 CLS #1: GOSUB 1000
570 FOR x=0 TO count-1
580 POKE buffer+(x*3+1),envelope(x*3+1): skip%=envelope(x*3+
1)
590 p$=STR$(envelope(x*3+1)): GOSUB 900
600 POKE buffer+(x*3+2),envelope(x*3+2): size%=envelope(x*3+
2)
610 pp=envelope(x*3+2): IF pp)127 THEN pp=pp-256
620 p$=STR$(pp): GOSUB 900
630 POKE buffer+(x*3+3),envelope(x*3+3): pause%=envelope(x*3
+3)
640 p$=STR$(envelope(x*3+3)): GOSUB 900
650 NEXT x
660 length%=length%+skip%*pause%
670 CALL routine
680 IF VV=0 THEN GOSUB 910: RETURN
690 off%=size%*10: IF size%)127 THEN off%=(size%-256)*10
700 FOR x=1 TO skip%
710  soundy=(soundy+off%) MOD 150: IF soundy<0 THEN soundy=1
50-soundy
720   FOR c=1 TO (10/51)*pause%+2: soundx=soundx+(sc%)
730    DRAW soundx+21,soundy+211,2+nk
740   NEXT
750 NEXT
760 DRAW soundx+21,soundy+211,2+nk
770 RETURN
780 REM redraw graph
790 LOCATE 5,20: PRINT CHR$(18);: INPUT "Section";sec%: IF s
ec%<1 OR sec%)5 THEN 790
800 count=sec%: GOSUB 360
810 CLS: soundx=0: soundy=INT(sv%*9.7): length%=0: GOSUB 270
820 FOR section=1 TO 5
830   count=section: GOSUB 540
840 NEXT
850 GOSUB 870
860 RETURN
870 REM sound
880 SOUND 1,tt,length%,sv%,1: RETURN
890 SOUND 1,tt,length%,sv%,,1: RETURN
900 PRINT #1,MID$(P$,(2-ABS(VAL(p$)(0)));",";: RETURN
910 FOR x=1 TO skip%
920  soundy=soundy-((15/127)*ss): IF soundy<0 THEN soundy=0
930 IF soundy)189 THEN soundy=189
940  FOR c=1 TO (7/51)*pause%+2: soundx=soundx+sc%
950   DRAW soundx+21,soundy+211,2+nk
960  NEXT
970 NEXT
980 DRAW soundx+21,soundy+211,2+nk
990 RETURN
1000 IF VV=1 THEN PRINT #1, "ENV 1, "; ELSE PRINT #1, "ENT 1
, ";
1010 RETURN
```

SPEAKING SPECTRUM SPANISH

Using your home micro to help you learn a foreign language seems to be a popular pastime – judging by the great range of software available for most machines. Bobby Pickering compared the relative merits of two Spanish packages for the Spectrum, and picked up a basic vocabulary in the process

The Spanish Tutor (Kosmos, £9.95) comprises two self-contained cassettes – Level A and Level B – each having a control program and a range of vocabulary lessons. On the Level A cassette, these include Foods, Parts of the body, Numbers and so on; the Level B cassette includes more advanced vocabulary, grouping words in their word classes – such as Verbs and Adjectives. Once a lesson has been selected and loaded from cassette, the program runs through the vocabulary, giving each word in both languages and expecting you to commit them to memory. For example, the first entry in the Colours lesson is 'red' and its Spanish equivalent 'rojo'; these are displayed on the screen, and when you're ready to go onto the next entry you press the Space bar.

Once you've memorised all the words in the vocab lesson, then you can take advantage of the test facility. Words in the lesson are displayed in one language and you must type in their equivalent in the other language. If you type in a wrong letter, the computer emits a beep. There are further options, allowing you to create, save and verify your own lessons, which suggests that the tape is aimed at language teachers who can use it as a framework on which to build further, perhaps more advanced, lesson files.

The tapes have a large combined vocabulary, contain useful features such as special characters and colour-coding of masculine and feminine words, and have room for creating a vocab database of your own. But in some respects they are of little use to people who want to study independently to acquire a basic speaking knowledge of the language (such as people who are going on holiday in Spain). The package particularly lacks any indication of pronunciation (how do you actually say 'rojo'?), and this is a major weakness.

Spanish Tutor is marketed by Kosmos Software, 1 Pilgrims Close, Harlington, Beds LU5 6LX.

Linkword Spanish (Silversoft, £12.95), on the other hand, is an attractively packaged and well thought-out learning program. It's based on the Gruneberg Linkword Language System, devised by Dr Michael Gruneberg, a Welsh psychology lecturer. The system aims to aid the learning process by making associative links between the corresponding words. The Spanish word for 'business' is 'negocio'; the system thus asks you to imagine making a 'business negotiation'. If the two words are similar – 'taxi', for example, is 'taxi' – then you are asked to use the keyword 'bullfighter' in the association. Picture a 'taxi full of bullfighters' says the program, and leaves the rest to your imagination. At first glance this may all seem a little silly, but it's a system that produces good results.

The package contains two cassettes – one containing the computer program (which is divided into 10 separate lessons) and a corresponding audio cassette (broken into 10 equivalent sections). The two cassettes go hand in hand and provide a complete independent learning package.

Accompanying the cassettes is a clearly laid-out instruction booklet, which includes a complete glossary of all the terms used in the program. At the end of the audio cassette, a few of the differences between Castillian Spanish (used in the program) and South American Spanish are mentioned. But, as pointed out, these are differences of pronunciation that are equivalent to the differences found between English speakers in North America and the UK. The program doesn't attempt to delve into the intricacies of grammar; nor does it give the correct accents on the Spanish words ('due to technical considerations' according to the inlay). Nevertheless it is a superb way to get a basic grounding in the local lingo before booking your flight to Torremolinos or the Costa del Sol.

Linkword Spanish is marketed by Silversoft Ltd, London House, 271/273 King Street, London W6.